The M-calculus: a Higher-Order Distributed Process Calculus

Alan Schmitt, Jean-Bernard Stefani

▶ **To cite this version:**

Alan Schmitt, Jean-Bernard Stefani. The M-calculus: a Higher-Order Distributed Process Calculus. [Research Report] RR-4361, INRIA. 2002. inria-00072227

HAL Id: inria-00072227
https://inria.hal.science/inria-00072227

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The M-calculus: a higher-order distributed process calculus

Alan Schmitt, Jean-Bernard Stefani

N° 4361

January 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4361--FR+ENG

INRIA
RHÔNE-ALPES

The M-calculus: a higher-order distributed process calculus

Alan Schmitt, Jean-Bernard Stefani

Thème 1 — Réseaux et systèmes
Projets MOSCOVA – SARDES

Rapport de recherche n° 4361 — January 2002 — 42 pages

Abstract: This report presents a new distributed process calculus, called the M-calculus. Key insights for the calculus are similar to those laid out by L. Cardelli for its calculus of ambients. Mobile Ambients and other recent distributed process calculi such as the Join calculus or the Dπ-calculus introduce notions of distributed locations or localities, corresponding to a spatial partitioning of computations and embodying different features of distributed computations (e.g. failures, access control, process migration, etc). However these calculi remain unsatisfactory in that they account for a single predefined behavior associated with a locality: in a large distributed system, localities may be of different types and exhibit a wide range of behaviors. This report tries to remedy to this limitation in defining a distributed programming model that allows the explicit programming of locality behavior. More precisely, the M-calculus can be understood as a generalization of the Join calculus and of G. Boudol's blue calculus that provides: distributed localities with programmable behavior (cells), higher-order processes, process mobility, and dynamic binding features.

Key-words: process calculus, distributed programming, programming model, mobile processes, π-calculus

Affi liations: Alan Schmitt, Projet MOSCOVA, INRIA Rocquencourt - Jean-Bernard Stefani, Projet SARDES, INRIA Rhônes-Alpes.

This research has been supported in part by the Marvel RNRT project.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Le M-calcul : un calcul de processus réparti d'ordre supérieur

Résumé : Ce rapport de recherche présente un nouveau calcul de processus réparti, appelé le M-calcul. Ce calcul constitue un modèle formel de programmation répartie avec mobilité de processus, dont les motivations sont similaires à celles avancées par L. Cardelli pour son calcul des ambients. Le calcul des ambients et d'autres calculs de processus répartis comme le Join calcul ou le Dπ-calcul, introduisent des notions de localités, qui correspondent à une partition spatiale des exécutions réparties et qui sont censées capturer différents aspects de ces exécutions (par exemple, aspects liés aux défaillances, au contrôle d'accès, à la migration de processus, etc). Ces différents calculs de processus demeurent cependant insatisfaisants car ils associent tous un comportement unique et prédéfini à la notion de localité qu'ils introduisent. Dans un système réparti de grande taille, on peut au contraire s'attendre à trouver des localités de différentes sortes, avec des comportements très variés. Le M-calcul essaie de pallier à cette limitation en fournissant un modèle de programmation réparti qui autorise la programmation explicite du comportement des localités. Plus précisément, le M-calcul peut être compris comme une généralisation à la fois du Join calcul et du calcul bleu de G. Boudol qui comprend : des localités au comportement programmable (cellules), des processus d'ordre supérieur, de la mobilité de processus et une forme de liaison dynamique.

Mots-clés : calcul de processus, modèle de programmation, programmation répartie, mobilité de processus, π-calcul.

Contents

Chapter 1

Introduction

We present in this paper a new process calculus, called the M-calculus, which represents an attempt at defining a formal distributed programming model. Key insights for the calculus are similar to those laid out in [7], where Luca Cardelli argues that large scale, WAN-based, distributed programming is substantially different from LAN-based distributed programming. When designing a wide-area distributed system, one can no longer ignore the different forms of physical and logical barriers that structure and partition the system. Such barriers arise because of three main reasons:

- The spatial extent of the system and its wide-area communication topology, which make communication delays and the potential occurrence of (network or node) failures difficult or impossible to mask, thereby imposing on distributed system designers and programmers a spatial partitioning of the system into distinct regions, characterized by different failure modes, interaction costs, and accessibility.
- The logical partitioning of the system into different administrative domains, under the control of independent authorities that operate their respective domains with different policies concerning interconnection, quality of service, security, fault management, etc.
- The presence of mobile objects in the system, be they people, computers, or software entities, which make disconnected operation the rule and which require an explicit handling of locations both for interacting with a mobile object, and for operating it.

These insights led L. Cardelli to the design of the Ambient calculus [6], which incorporates basic primitives for handling barriers (creating, opening, and entering a region protected by a barrier). The Ambient calculus has several deficiencies (notably, grave forms of concurrent interferences [17] and atomicity conditions for key primitives that make its implementation difficult [14]). It also suffers from not adequately accounting for the variety of barriers that may occur in a large scale distributed setting. Several other distributed process calculi, based on notions of locations or localities [8], have tried to address them. For instance, Nomadic Pict [22] provides process mobility; the π_{1l}-calculus [1] and the distributed join-calculus with failures [13] embody both process mobility and failure detection; other calculi, such as the Dπ-calculus [16] and KLAIM [9], provide both process mobility and access control.

In our view, these calculi remain unsatisfactory: while they may succeed in capturing key features of distributed locations, they usually account for one kind of location only. Instead, one would expect a distributed process calculus to allow for the modelling of different forms of distributed locations, each one possibly exhibiting a different sort of behavior, e.g. with respect to failure modes, access control, extensibility and reconfiguration. The calculus introduced in this paper is a first step in that direction. More precisely, the M-calculus can be understood as a generalization of the distributed join calculus and of the blue calculus [4, 26] that provides: distributed localities with explicit, programmable behavior (cells); higher-order processes; and dynamic binding features.

1.1 Requirements for a distributed programming model

Before describing the main features of the M-calculus, we gather in in this section key informal *requirements* that a distributed process programming model should satisfy.

A first requirement for a distributed programming model concerns its "implementability". A programming model should provide primitives which can be implemented reasonably efficiently to serve as a basis for concrete and usable programming languages. We capture this as:

Requirement 1 *A distributed programming model should provide primitive constructs that closely mirror typical communication and configuration capabilities available in distributed environments such as the Internet.*

There are two main reasons behind this requirement:

- Distributed application programmers should have an unbiased view of the basic costs and trade-offs that need to be made when building distributed applications. In particular, basic communication costs and the possible occurrence of failures should not be masked by distributed programming model primitives[1].
- Certain application domains, and "system-level" programming, require access to low-level system constructs. For instance, system management applications require access to system-level and network-level resources to do their job. This is at odds with the introduction of too high-level abstractions in the programming model, which would mask such low-level constructs.

This requirement has several consequences in terms of constructs to be introduced in a distributed programming model. As explained above, a large distributed system should be understood primarily as a collection of distributed locations or regions, with distinct behaviors. We shall call *cells* such locations or regions.

Cells come in many varieties, for instance:

- resource containers, encompassing hardware and software resources under the control of a single resource manager (e.g. information processing nodes in a computer network);
- processes, delineating address spaces dedicated to the execution of programs written in a single programming language (e.g. operating system processes);
- failure zones, encompassing entities that may fail together, according to certain failure modes (e.g. fail silent machines);
- administrative domains, encompassing computing resources under the control and management of a single authority (e.g. network management domains);
- security regions, corresponding to sets of nodes controlled by security policies and firewalls;
- spatial locations, encompassing entities located at a given address in a computer network.

Notions of ambients in the Ambient calculus [6], of seals in the Seal calculus [25], of localities in the Join calculus [11], in the π_{1l}-calculus [1], and in the Dπ-calculus [16], correspond to variants of this general notion of *cell*. In these calculi, cells are used to make explicit the spatial structure of computations (as flat or tree-shaped sets of domains). They differ in the (implicit) behavior they attach to their respective notion of cell. However, all these calculi adopt a homogeneous view of cells, each cell being capable of a single pre-defined behavior (e.g. with respect to failure or process migration). In a large distributed system, we can expect cells of various kinds to coexist. This leads us to the following requirement:

[1] If necessary, appropriate abstractions, e.g. for fault-tolerance or multi-party communication, can be built on top of the model's primitives, and made available as *libraries* to application programmers.

Requirement 2 *A distributed programming model should include, as a primitive construct, a notion of cell, understood as a means to spatially partition a distributed computation, and should allow the definition of arbitrary domain behaviors.*

From the examples given above, it appears that a cell can be characterized by two elements:

- a set of entities belonging to the interior of the cell, which we shall call the *plasm* of the cell;
- a controlling entity, that embodies the barrier implemented by a cell, which we shall call the *membrane* of the cell.

We now consider what general requirements apply to these elements. First, the membrane of a cell can play the role of a filter with respect to messages which are sent to the cell's plasm. This type of behavior is manifest, for instance, in firewalls (membranes for security domains), or in so-called component containers such as e.g. in EJB (Enterprise Java Beans) [24] or CORBA Components [20]. Modeling network nodes with their protocol machinery, at various levels of abstractions, also requires the introduction of cells and of their associated membranes, capable of intercepting and processing incoming and outgoing protocol data units. We record this as:

Requirement 3 *A distributed programming model should allow the definition of cell membranes that can intercept and process messages which are going to, or coming from, their associated cell plasms, possibly changing their own state in the process.*

Another requirement is for cell membranes to evolve by receiving and sending messages to their environment. System management applications, for instance, typically require access to sets of managed objects (e.g. for querying the state of a given machine, for shutting it down, for activating it, etc). These applications further illustrate the fact that state changes of a cell membrane may cause correlative changes in the associated cell plasm. For instance, shutting down a given machine, understood as both a resource cell and a failure cell, will cause the different computations it supports to be terminated (and, perhaps, moved to different storage cells in a "passive" state). We record this as:

Requirement 4 *A distributed programming model should allow the definition of cell membranes that can send and receive messages, possibly changing their own state and causing state changes in their associated plasm.*

Another requirement pertains to the creation of new cells and to the migration of plasms between cells. Dynamic reconfiguration in an open distributed system implies the ability to introduce new objects and new subsystems, e.g. to increase the capacity of the system under a changing load, or to update parts of the system with new (hardware or software) technology. Modelling mobile systems, i.e. systems with physically mobile subsystems (portable PCs, PDAs, mobile phones, etc) or mobile software objects (e.g. mobile agents), implies the ability to move objects between different spatial locations. This translates into the following requirement:

Requirement 5 *A distributed programming model should allow the dynamic creation of new cells, and should provide the ability to move all, or part of, a cell plasm from one cell to another.*

These requirements are not adequately covered by current distributed process calculi (see [8] and [27] for recent overviews). For instance, the distributed join calculus partially satisfies requirement 1, but fails to satisfy 3 and 4, and only very partially satisfies 2, and 5. The Ambient calculus does not satisfy requirement 1 as demonstrated e.g. in [14], does not satisfy 3 and 4, and satisfies only very partially requirements 2 and 5.

1.2 Introducing the M-calculus

The programming model described in this report, the M-calculus, takes the form of a process calculus which extends both the blue calculus ([4],[26]) and the join calculus ([10, 11, 13]).

The blue calculus is interesting for the present study because of its elegant mix of the λ-calculus and of the π-calculus. From the blue calculus, we have retained:

1. a direct embedding of the λ-calculus;
2. the idea that messages are functional applications;
3. the separation between (lambda) abstractions and declarations.

The join calculus is interesting because of its notion of location, which we extend with our notion of cell, and its implementability in a distributed setting. From the join calculus, we have retained:

1. message patterns associated with definitions, i.e. the possibility to specify synchronization patterns in the form of finite sets of messages;
2. named cells bearing a unique name and organized as a tree, with automatic routing of messages addressed to a cell.

One of the main issues in deriving an effective programming model based on the notion of cell has to do with the expression of control in a cell, i.e. how to express control abilities of a cell membrane P in a cell $a(P)[Q]$. There are at least two different ways one could consider for expressing that control:

1. One could make use of synchronization primitives similar to the ones used in the Meije-calculus [3], e.g. using a combination of triggering and restriction to implement a form of regulative superimposition [15].
2. Alternatively, one could exploit reflection ideas, namely, by considering each cell membrane P as a form of meta-object in the cell construct $a(P)[Q]$.

In this report, we adopt a hybrid approach. This leads us to introduce an explicit operator for the passivation of processes contained in a cell plasm, and to allow for explicit message exchanges between cell membrane and cell plasm.

In a departure from both the blue calculus and the join calculus we consider a higher-order value passing calculus, where application is not restricted to simple names but can be applied to arbitrary processes.

In the M-calculus, cells are organized in a tree structure (similarly to locations in the join-calculus). This choice is a simplification over the perceived organization of cells in an real distributed system, which may overlap and organize themselves in an arbitrary graph structure.

In the M-calculus, routing of messages between cells is automatic but not transparent: messages are sent toward the destination cell, but when they cross a cell boundary, the cell membrane is given the opportunity to intercept the message. This allows cell membranes in the M-calculus to retain full control over communications. However, transparent routing is very easy to encode, providing, within the same calculus, the two forms of communication provided by the two-level architecture of Nomadic Pict [22].

A last feature of the M-calculus consists in the ability to perform a form of dynamic binding, an essential feature for practical distributed programming.

Chapter 2

The M-calculus: syntax and operational semantics

We define in this chapter the syntax and semantics of the M-calculus, together with a simple type system enforcing the unicity of cell names.

2.1 Syntax

The syntax of the M-calculus is given in Figures 2.1 and 2.2. We postulate an infinite countable set of *cell names*, CN (with a distinguished element ϵ), an infinite denumerable set of resource names, Ref (with three distinguished elements, $\mathbf{i}$, $\mathbf{o}$ and $\mathbf{e}$), and an infinite denumerable set of variables, Var. The set of names, N, is the disjoint union of CN, Var, Ref, and the set of addressed resource names ($\mathsf{CN} \times \mathsf{Ref}$). We let r and its decorated variants range over Ref; x, y and their decorated variants range over Var; a, b and their decorated variants range over CN; u, v and their decorated variants range over N. We let n and its decorated variants range over the set of resolved names, that is the disjoint union of CN and Ref. We let θ and its decorated variants range over the set of finite subtitutions over names. We note $\theta = \{t_1/u_1, \ldots, t_q/u_q\}$ a finite substitution where the term t_i is substituted to the name u_i. We note $P\theta$ or $P\{t_1/u_1, \ldots, t_q/u_q\}$ the image of the term P under substitution θ.

We let P, Q, R, S and their decorated variants range over processes of the M-calculus. Similarly to the λ-calculus, certain M-calculus processes are seen as *values*. Values are given by V in Figure 2.1. Values are either the null value $()$, names, or λ-abstractions (processes of the form $\lambda x.P$). We use V, and its decorated variants to range over values. Processes of the form $\langle D \rangle$ are called *definitions*. They correspond to replicated resources in the blue calculus and to definitions in the join calculus. We let $\langle D \rangle$ and its decorated variants range over definitions. We let J and its decorated variants range over message patterns.

In a process $\lambda x.P$ or $\nu n.P$, the scope extends as far to the right as possible. We use standard abbreviations from the λ-calculus and the π-calculus: $PQ_1 \ldots Q_n$ for $(\ldots(PQ_1)\ldots Q_n)$, and $\lambda u_1 \ldots u_q.P$ for $\lambda u_1 . \ldots \lambda u_q.P$. Similarly we use $\nu u_1 \ldots u_q.P$ for $\nu u_1 . \ldots \nu u_q.P$. We use $\widetilde{t}$ to denote finite vectors of terms $(t_1, \ldots, t_q)$. When the sizes of vectors $\widetilde{t}$ and $\widetilde{u}$ match (i.e. $\widetilde{t} = (t^1, \ldots, t^p)$ and $\widetilde{u} = (u^1, \ldots, u^p)$), we note $\{\widetilde{t}/\widetilde{u}\}$ the substitution $\{t^1/u^1, \ldots, t^p/u^p\}$. The same notational convention applies to vectors of vectors. We also make use of the notation $\lambda.P$ to stand for $\lambda x.P$, with x not free in P.

The informal meaning of the different constructs of the M-calculus is as follows:

- At the top-level, an M-calculus program takes the form of a root cell, $\epsilon[P]$. ϵ is the name of the root cell, P is an arbitrary M-calculus process. Note that, from a behavioral point of view, the root cell $\epsilon[P]$ is really equivalent to a cell of the form $\epsilon(\mathsf{0})[P]$, i.e. a cell with a null membrane.

- $\mathbf{0}$ is the empty process, which has no associated transition, and is a neutral element for the parallel composition operator, $|$.
- A process of the form $a(P)[Q]$ is called a cell. Each cell $a(P)[Q]$ consists of a *cell name* a, a membrane process P, and a plasm process Q. A membrane process and a plasm process may in turn be formed of a parallel composition of cells. Thus, an M-calculus program in fact consists in a forest of cells executing in parallel.
- A process that is a value V is either the null value $()$, a name u, or a lambda abstraction $\lambda x.P$. A name, as defined in Figure 2.2, can be either a variable, a resource name (i.e. a name defined in a join pattern of a definition), a cell name, or an addressed resource name (i.e. a concatenation of a cell name and of a resource name). A lambda abstraction $\lambda x.P$ is considered aa a value as there is no evaluation taking place inside a lambda abstraction. Note that a lambda abstraction operates on a variable, not on a resource name or a cell name. Thus, a process of the form $\lambda x.\langle x = P\rangle$ or $\lambda x.x(P)[Q]$is not allowed in the M-calculus. This constraint is directly inspired by a similar constraint in the blue calculus and is motivated in [4].
- A process of the form $(P \mid Q)$ is a parallel composition of two processes P and Q. The parallel operator is associative, commutative and has $\mathbf{0}$ as its neutral element.
- A process of the form PQ is an application: process P is a function which is applied to argument Q. Q must evaluate to a value V for the application to reduce.
- The operator $\nu n.$ is a restriction operator similar to that of the π-calculus [18].
- The construct $[s = V]P, Q$ is a standard conditional branch. If V is equal to s, then it evaluates to P, otherwise it evaluates to Q.
- $\langle J_1 = P_1; \ldots ; J_q = P_q\rangle$ denotes a resource definition. Intuitively, this process responds to the occurrence of a message pattern J_l with the firing of process P_l, instantiated with values carried by the actual messages in the pattern. A message pattern is defined by a finite parallel composition of messages. Messages in the M-calculus take the form of applications $rV_1 \ldots V_q$ where the head term r is a resource name.
- An expression of the form $\mathtt{pass}_a\ V$ in the membrane of the cell named a, causes the whole cell to be passivated and the function V to be applied to the results of that passivation (the passivated cell membrane and the passivated cell plasm). Passivation consists in placing a process under a lambda abstraction.

2.2 Operational semantics

An M-context is a term built according to the same grammar than for standard M-calculus terms, plus a constant $\cdot$, the hole. We use $P\{\cdot\}$ to denote M-contexts. Filling the hole in $P\{\cdot\}$ with an M-calculus term Q results in an M-calculus term noted $P\{Q\}$. Evaluation contexts in the M-calculus are M-contexts built according to the grammar given in Figure 2.3.

We denote by $\mathsf{fn}(P)$ the set of free names of P. The set $\mathsf{fn}(P)$ is defined recursively in Figure 2.4. We denote by $\mathsf{df}(J)$ the set of defined names of the join pattern J, which is defined as

$$\mathsf{df}(r_1\widetilde{x}_1 \mid \ldots \mid r_q\widetilde{x}_q) = \{r_1, \ldots , r_q\}$$

We denote by $\mathsf{dln}(P)$ the set of defined local names of P. The set $\mathsf{dln}(P)$ is defined recursively in Figure 2.5. We denote by $\mathsf{cells}(P)$ the set of the names of active cells. The set $\mathsf{cells}(P)$ is defined recursively in Figure 2.6.

$\mathcal{S}$	::=		Top-level configuration
		$\epsilon[P]$	root cell
	|	$\nu n.\mathcal{S}$	restriction
P	::=		Process
		$\mathbf{0}$	inert process
	|	V	value
	|	$a(P)[P]$	cell
	|	$(P \mid P)$	parallel composition
	|	(PP)	application
	|	$\nu n.P$	restriction
	|	$([s = V]P, P)$	name testing
	|	$\langle D \rangle$	definition
	|	$\mathtt{pass}_a\ V$	passivation operator
V	::=		Value
		$()$	void
	|	u	name
	|	$\lambda x.P$	lambda abstraction
D	::=		Definition
		$\bot$	empty definition
	|	$J = P$	reaction rule
	|	$D; D$	composition
J	::=		join pattern
		$r\widetilde{x}$	message
	|	$J \mid J$	join

Figure 2.1: Syntax

s	::=		service name
		r	resource name
	|	$a.r$	addressed resource name
n	::=		resolved name
		r	resource name
	|	a	cell name
u	::=		name
		a	cell name
	|	x	variable
	|	s	service name

Figure 2.2: Names

$\mathbf{E}$	$::=$		evaluation context
		$\cdot$	hole
	$\mid$	$(\mathbf{E}V)$	function
	$\mid$	$(P\mathbf{E})$	argument
	$\mid$	$\nu n.\mathbf{E}$	restriction
	$\mid$	$(\mathbf{E} \mid P)$	parallel
	$\mid$	$a(P)[\mathbf{E}]$	plasm
	$\mid$	$a(\mathbf{E})[P]$	membrane
	$\mid$	$\epsilon[\mathbf{E}]$	top

Figure 2.3: Evaluation Contexts

$$\mathsf{fn}(()) = \emptyset \qquad \mathsf{fn}(0) = \emptyset$$
$$\mathsf{fn}(u) = \{u\} \quad \mathsf{fn}(a.r) = \{a, r\} \qquad \mathsf{fn}(\lambda x.P) = \mathsf{fn}(P) \setminus \{x\}$$
$$\mathsf{fn}(\nu n.P) = \mathsf{fn}(P) \setminus \{n\} \qquad \mathsf{fn}(PQ) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$$
$$\mathsf{fn}(\langle D \rangle) = \mathsf{fn}(D) \qquad \mathsf{fn}(D; D') = \mathsf{fn}(D) \cup \mathsf{fn}(D')$$
$$\mathsf{fn}(\bot) = \emptyset \qquad \mathsf{fn}(J = P) = (\mathsf{fn}(P) \setminus \mathsf{fn}(J)) \cup \mathsf{df}(J)$$
$$\mathsf{fn}(r_1\widetilde{x}_1 \mid \ldots \mid r_q\widetilde{x}_q) = \mathsf{fn}(r_1\widetilde{x}_1) \cup \ldots \cup \mathsf{fn}(r_q\widetilde{x}_q) \qquad \mathsf{fn}(r\widetilde{x}) = \{r, x_1, \ldots, x_p\} \quad \widetilde{x} = (x_j)_{j \in \{1,\ldots,p\}}$$
$$\mathsf{fn}(a(P)[Q]) = \{a\} \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q) \qquad \mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$$
$$\mathsf{fn}([s = V]P, Q) = \mathsf{fn}(s) \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q) \cup \mathsf{fn}(V) \qquad \mathsf{fn}(\mathtt{pass}_a\ V) = \mathsf{fn}(V)$$
$$\mathsf{fn}(\nu n.\mathcal{S}) = \mathsf{fn}(\mathcal{S}) \setminus \{n\} \qquad \mathsf{fn}(\epsilon[P]) = \mathsf{fn}(P)$$

Figure 2.4: Free names

$$\mathsf{dln}(()) = \emptyset \qquad \mathsf{dln}(0) = \emptyset$$
$$\mathsf{dln}(u) = \emptyset \qquad \mathsf{dln}(\lambda x.P) = \emptyset$$
$$\mathsf{dln}(\nu n.P) = \mathsf{dln}(P) \setminus \{n\} \qquad \mathsf{dln}(PQ) = \emptyset$$
$$\mathsf{dln}(\langle D \rangle) = \mathsf{dln}(D) \qquad \mathsf{dln}(D; D') = \mathsf{dln}(D) \cup \mathsf{dln}(D')$$
$$\mathsf{dln}(\bot) = \emptyset \qquad \mathsf{dln}(J = P) = \mathsf{dln}(J)$$
$$\mathsf{dln}(r_1\widetilde{x}_1 \mid \ldots \mid r_q\widetilde{x}_q) = \{r_1, \ldots, r_q\} \qquad \mathsf{dln}(a(P)[Q]) = \emptyset$$
$$\mathsf{dln}(P \mid Q) = \mathsf{dln}(P) \cup \mathsf{dln}(Q) \qquad \mathsf{dln}([s = V]P, Q) = \emptyset$$
$$\mathsf{dln}(\mathtt{pass}_a\ V) = \emptyset \qquad \mathsf{dln}(\mathcal{S}) = \emptyset$$

Figure 2.5: Defined local names

$$\mathsf{cells}(()) = \emptyset \qquad \mathsf{cells}(0) = \emptyset$$
$$\mathsf{cells}(u) = \emptyset \qquad \mathsf{cells}(\lambda x.P) = \emptyset$$
$$\mathsf{cells}(\nu n.P) = \mathsf{cells}(P) \setminus \{n\} \qquad \mathsf{cells}(PQ) = \emptyset$$
$$\mathsf{cells}(\langle D \rangle) = \emptyset \qquad \mathsf{cells}(a(P)[Q]) = \{a\} \cup \mathsf{cells}(P) \cup \mathsf{cells}(Q)$$
$$\mathsf{cells}(P \mid Q) = \mathsf{cells}(P) \cup \mathsf{cells}(Q) \qquad \mathsf{cells}([s = V]P, Q) = \emptyset$$
$$\mathsf{cells}(\mathtt{pass}_a\ V) = \emptyset \qquad \mathsf{cells}(\nu n.\mathcal{S}) = \mathsf{cells}(\mathcal{S}) \setminus \{n\}$$
$$\mathsf{cells}(\epsilon[P]) = \mathsf{cells}(P)$$

Figure 2.6: Active cells

$$\frac{n \notin \mathsf{fn}(Q)}{(\nu n.P) \mid Q \equiv \nu n.(P \mid Q)}\ [\text{STRUCT.NU.PAR}] \qquad \frac{}{\epsilon[\nu n.P] \equiv \nu n.\epsilon[P]}\ [\text{STRUCT.NU.TOP}]$$

$$\frac{n \notin \mathsf{fn}(Q) \ \wedge \ n \neq a}{a(\nu n.P)[Q] \equiv \nu n.a(P)[Q]}\ [\text{STRUCT.NU.MEM}] \qquad \frac{n \notin \mathsf{fn}(P) \ \wedge \ n \neq a}{a(P)[\nu n.Q] \equiv \nu n.a(P)[Q]}\ [\text{STRUCT.NU.PLASM}]$$

$$\frac{P =_\alpha Q}{P \equiv Q}\ [\text{STRUCT.}\alpha] \qquad \frac{P \equiv Q}{\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}}\ [\text{STRUCT.CONTEXT}]$$

Figure 2.7: Structural equivalence

Equivalence of two processes P and Q up to α-conversion is noted $P =_\alpha Q$. We recall that in $\nu n.P$, $\lambda x.P$, and $r_1\widetilde{x_1} \mid \ldots \mid r_q\widetilde{x_q} = P$, the names and variables n, x, and each x_i are bound in P.

The operational semantics of the M-calculus is defined in the CHAM style [2], using a structural equivalence, $\equiv$, and a reduction relation, $\rightarrow$.

The structural equivalence, $\equiv$, is the smallest equivalence relation that satisfies the rules given in Figure 2.7, where operator "|" for processes, and operator ";" for definitions are taken to be commutative and associative, with $\mathsf{0}$ and $\perp$ as their neutral elements, respectively.

The intuitive meaning of these rules is as follows:

- Rules STRUCT.NU.PAR, STRUCT.NU.TOP, STRUCT.NU.MEM, and STRUCT.NU.PLASM are scope extrusion rules. They stipulate that the restriction operator creates new names which are unique in a whole configuration.
- Rule STRUCT.α asserts that α-equivalent processes are equivalent.
- Rule STRUCT.CONTEXT asserts that equivalence $\equiv$ is a congruence for evaluation contexts.

The reduction relation for the M-calculus, $\rightarrow$, is defined as the smallest relation that satisfies the rules given in Figures 2.8, 2.9 and 2.10.

The intuitive meaning of these rules is as follows:

- Rule RED.BETA is the standard beta-reduction rule of the λ-calculus.
- Rules RED.IF.THEN and RED.IF.ELSE are standard rules for a conditional branch.
- Rule RED.RES specifies the behavior of a resource when it receives a set of messages that matches one of its join patterns: a new instance of the resource process is instantiated with its formal parameters bound to the arguments of the messages. This rule closely mirrors the behavior of definitions in the join-calculus.
- Rule RED.CONTEXT indicates that reductions are possible within an evaluation context, i.e. inside a cell construct (in its membrane or its plasm), inside branches of a parallel composition, and under a restriction operator. By contrast, reduction is not possible under a λ-abstraction.
- Rules RED.PROC.EQUIV and RED.TOP.EQUIV stipulates that the set of reductions is the same for equivalent processes or top-level configurations.
- Rules in Figure 2.9 specify how routing of a message bearing a resource name is effected. Notice in particular that, apart from rules RED.MEM.MESS.OUT and RED.PLASM.MESS.OUT, there are no rules

$$\frac{}{(\lambda x.P)V \to P\{^V/_x\}}\ [\text{RED.BETA}] \qquad \frac{}{([n = n]P, Q) \to P}\ [\text{RED.IF.THEN}]$$

$$\frac{n \neq V}{([n = V]P, Q) \to Q}\ [\text{RED.IF.ELSE}] \qquad \frac{}{a(\mathtt{pass}_a\ V \mid P)[Q] \to V(\lambda.P)(\lambda.Q)}\ [\text{RED.PASSIV}]$$

$$\frac{\langle D\rangle = \langle D_0\ ;\ r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} = P\rangle}{\langle D\rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} \to \langle D\rangle \mid P\{\widetilde{V_i}/\widetilde{x_i}\}}\ [\text{RED.RES}] \qquad \frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}}\ [\text{RED.CONTEXT}]$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\ [\text{RED.PROC.EQUIV}]$$

$$\frac{\mathcal{S}_1 \equiv \mathcal{S}_1' \qquad \mathcal{S}_1' \to \mathcal{S}_2' \qquad \mathcal{S}_2' \equiv \mathcal{S}_2}{\mathcal{S}_1 \to \mathcal{S}_2}\ [\text{RED.TOP.EQUIV}]$$

Figure 2.8: Reduction: Computing Rules

$$\frac{r \notin \mathsf{dln}(P) \qquad r \in \mathsf{dln}(Q)}{a(P \mid r\widetilde{V})[Q] \to a(P)[Q \mid r\widetilde{V}]}\ [\text{RED.MESS.PLASM.IN}]$$

$$\frac{r \in \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{a(P)[Q \mid r\widetilde{V}] \to a(P \mid r\widetilde{V})[Q]}\ [\text{RED.MESS.PLASM.OUT}]$$

$$\frac{r \notin \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{a(P)[Q \mid r\widetilde{V}] \to a(P \mid \mathbf{o}(\lambda.r\widetilde{V}))[Q]}\ [\text{RED.MESS.FILTER.OUT}]$$

$$\frac{r \notin \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{b(a(P \mid r\widetilde{V})[Q] \mid R)[S] \to b(a(P)[Q] \mid r\widetilde{V} \mid R)[S]}\ [\text{RED.MEM.MESS.OUT}]$$

$$\frac{r \notin \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{b(R)[a(P \mid r\widetilde{V})[Q] \mid S] \to b(R)[a(P)[Q] \mid r\widetilde{V} \mid S]}\ [\text{RED.PLASM.MESS.OUT}]$$

$$\frac{r \notin \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{\epsilon[a(P \mid r\widetilde{V})[Q] \mid R] \to \epsilon[a(P \mid \mathbf{e}(\lambda.r\widetilde{V}))[Q] \mid R]}\ [\text{RED.MESS.ERR}]$$

Figure 2.9: Reduction: Routing Rules (Part 1)

$$\frac{r \notin \mathsf{dln}(P) \qquad r \in \mathsf{dln}(Q)}{a.r\widetilde{V} \mid a(P)[Q] \to a(P \mid \mathbf{i}(\lambda.r\widetilde{V}))[Q]} \text{ [RED.ADDR.FINAL.PLASM]}$$

$$\frac{r \in \mathsf{dln}(P) \vee r \notin \mathsf{dln}(Q)}{a.r\widetilde{V} \mid a(P)[Q] \to a(P \mid r\widetilde{V})[Q]} \text{ [RED.ADDR.FINAL.MEM]}$$

$$\frac{}{a(P \mid a.r\widetilde{V})[Q] \to a(P \mid r\widetilde{V})[Q]} \text{ [RED.ADDR.MEM]}$$

$$\frac{}{a(P)[Q \mid a.r\widetilde{V}] \to a(P)[Q \mid r\widetilde{V}]} \text{ [RED.ADDR.PLASM]}$$

$$\frac{b \in \mathsf{cells}(P) \qquad b \neq a}{a(P)[Q] \mid b.r\widetilde{V} \to a(P \mid b.r\widetilde{V})[Q]} \text{ [RED.ADDR.MEM.IN]}$$

$$\frac{b \notin \mathsf{cells}(P) \qquad b \in \mathsf{cells}(Q) \qquad b \neq a}{a(P)[Q] \mid b.r\widetilde{V} \to a(P \mid \mathbf{i}(\lambda.b.r\widetilde{V}))[Q]} \text{ [RED.ADDR.FILTER.IN]}$$

$$\frac{b \notin \mathsf{cells}(P) \qquad b \in \mathsf{cells}(Q) \qquad b \neq a}{a(P \mid b.r\widetilde{V})[Q] \to a(P)[Q \mid b.r\widetilde{V}]} \text{ [RED.ADDR.PLASM.IN]}$$

$$\frac{b \notin \mathsf{cells}(P) \cup \mathsf{cells}(Q) \qquad b \neq a}{a(P \mid b.r\widetilde{V})[Q] \to a(P)[Q] \mid b.r\widetilde{V}} \text{ [RED.ADDR.MEM.OUT]}$$

$$\frac{b \notin \mathsf{cells}(Q) \qquad b \in \mathsf{cells}(P) \qquad b \neq a}{a(P)[Q \mid b.r\widetilde{V}] \to a(P \mid b.r\widetilde{V})[Q]} \text{ [RED.ADDR.PLASM.OUT]}$$

$$\frac{b \notin \mathsf{cells}(P) \cup \mathsf{cells}(Q) \qquad b \neq a}{a(P)[Q \mid b.r\widetilde{V}] \to a(P \mid \mathbf{o}(\lambda.b.r\widetilde{V}))[Q]} \text{ [RED.ADDR.FILTER.OUT]}$$

Figure 2.10: Reduction: Routing Rules (Part 2)

τ	::=		type
		Δ	process type
	|	σ	value type
s	::=		type scheme
		$\forall\widetilde{\alpha}\widetilde{\rho}.\sigma$	type scheme
σ	::=		value type
		`unit`	unit type
	|	α	type variable
	|	`dom`	cell name type
	|	$\sigma \to \tau$	function
Δ	::=		multiset of cell names
		$\emptyset$	empty multiset
	|	ρ	multiset variable
	|	a	cell name
	|	Δ, Δ	union

Figure 2.11: Types: Syntax

for a message bearing a resource name to enter a different cell. Such a message may only move up a tree of enclosing cells until it encounters the named resource in the plasm or in the membrane of a parent cell. If it fails to do so, an error message is generated, to be handled by the highest cell that the original message reached in the tree, as specified in rule RED.MESS.ERR. This last rule is the only one that applies when a message reaches a top-level cell.

- Rules in Figure 2.10 specify how routing of a message bearing an addressed resource name $b.r$ is executed. Roughly, routing is effected based on the cell name b. The message is transmitted unmodified until it reaches its destination cell. If the membrane does not contain the destination cell, the message is passed as an argument on the $\mathbf{i}$ port of the membrane that controls the plasm holding the target resource (rule RED.ADDR.FINAL.PLASM). Otherwise, it enters the cell membrane (rule RED.ADDR.FINAL.MEM). Note that messages that target cells inside or outside a particular cell plasm are sytematically filtered by the enclosing membrane, using the special $\mathbf{i}$ and $\mathbf{o}$ ports (rules RED.ADDR.FINAL.PLASM, RED.ADDR.FILTER.IN, RED.ADDR.FILTER.OUT).

2.3 Typing

The type system presented in this section enforces a property of unicity of names of active cells in a top-level configuration. The grammar for types is given in Figure 2.11. Intuitively, a process is typed by a multiset Δ that records the names of the active cells that appear in that process. At the top-level, it is required that each cell name appearing in the root-cell plasm occurs only once, i.e. that each cell in a top-level configuration has a unique name.

We use $\widetilde{\sigma}$ for vectors of types. We use $\forall\widetilde{\alpha}\widetilde{\rho}.\sigma$ for type schemes, where the type variables $\widetilde{\alpha}$ and the multiset variables $\widetilde{\rho}$ are generalized. We also extend the syntax by requiring new resource names to be anotated by their type scheme, as in $\nu r : s$. We use Δ and its decorated variants to denote multisets of cell names and multiset variables. The union operation between such multisets, ",", is the standard union

on multisets. The intersection operation between multisets, "$\cap$", is the standard intersection on multisets (taking the smallest number of occurrences in both multisets). The inclusion relation $\subseteq$ between multisets is also taken as the standard one.

By $\Delta - \Delta'$, we denote the multiset which is composed of the elements of Δ (cell names or multiset variables) after removing each element of Δ'. For instance $\rho, \rho, a, b, b - \rho, a, a, b = \rho, b$. We write $\Delta \setminus \Delta'$ for the multiset Δ where all occurences of any element of Δ' have been removed. We extend the "$\setminus$" operator on all types thus:

$$\begin{array}{rcl} \mathtt{unit} \setminus \Delta & = & \mathtt{unit} \\ \mathtt{dom} \setminus \Delta & = & \mathtt{dom} \\ \widetilde{\sigma} \setminus \Delta & = & \widetilde{(\sigma \setminus \Delta)} \\ (\sigma \to \tau) \setminus \Delta & = & (\sigma \setminus \Delta) \to (\tau \setminus \Delta) \end{array}$$

We define the symmetric $\wedge$ operator on types as follows:

$$\begin{array}{rcl} a, \Delta \wedge a, \Delta' & = & a, (\Delta \wedge \Delta') \\ \rho, \Delta \wedge \rho, \Delta' & = & \rho, (\Delta \wedge \Delta') \\ \Delta \wedge \Delta' & = & \Delta, \Delta' \text{ if } \Delta \cap \Delta' = \emptyset \\ \mathtt{unit} \wedge \mathtt{unit} & = & \mathtt{unit} \\ \mathtt{dom} \wedge \mathtt{dom} & = & \mathtt{dom} \\ \alpha \wedge \alpha & = & \alpha \\ \widetilde{\sigma} \wedge \widetilde{\sigma}' & = & \widetilde{(\sigma \wedge \sigma')} \text{ with tuples of identical size} \\ \sigma \to \tau \wedge \sigma \to \tau' & = & \sigma \to (\tau \wedge \tau') \end{array}$$

All other cases are undefined. As concerns multisets, $\Delta_1 \wedge \Delta_2$ is the multiset that contains for any name the maximum number of occurences of this name in Δ_1 and Δ_2.

We extend the "$\subseteq$" relation to types as follows:

$$\begin{array}{rcl} \mathtt{unit} \subseteq \mathtt{unit} & & \\ \mathtt{dom} \subseteq \mathtt{dom} & & \\ \alpha \subseteq \alpha & & \\ \widetilde{\sigma_i}^{i \in [1..n]} \subseteq \widetilde{\sigma'_i}^{i \in [1..n]} & \Leftarrow & (\sigma_i \subseteq \sigma'_i)^{i \in [1..n]} \\ \sigma \to \tau \subseteq \sigma \to \tau' & \Leftarrow & \tau \subseteq \tau' \end{array}$$

We use Γ and its decorated variants to denote type environments, i.e. finite mappings between names and types or type schemes.

We define the set of free set variables fsv as follows:

$$\begin{array}{rcl} fsv(\emptyset) & = & \emptyset \\ fsv(\rho) & = & \{\rho\} \\ fsv(a) & = & \emptyset \\ fsv(\Delta, \Delta') & = & fsv(\Delta) \cup fsv(\Delta') \end{array}$$

We extend fsv to types and type schemes, gathering all multiset variables that are not bound by a $\forall$. Similarly, we define free type variables ftv on types and type schemes as the set of type variables that are not bound by a $\forall$. We extend fsv and ftv on type environments in the trivial fashion, and we define fv as the union of ftv and fsv.

C	$::=$		context
		$\cdot : \tau$	hole for a process
	$\mid$	$\cdot$	hole for a definition
	$\mid$	$\cdot : \Delta$	hole for a top-level configuration
	$\mid$	$\epsilon[C]$	top-level cell
	$\mid$	$\nu r : s.C$	resource restriction
	$\mid$	$\nu a.C$	cell restriction
	$\mid$	$\lambda x.C$	function
	$\mid$	$a(C)[Q]$	cell membrane
	$\mid$	$a(P)[C]$	cell plasm
	$\mid$	$(C \mid P)$	left process parallel
	$\mid$	$(P \mid C)$	right process parallel
	$\mid$	$\mathtt{pass}_a\ C$	passivation
	$\mid$	(CQ)	left application
	$\mid$	(PC)	right application
	$\mid$	$([n = C]P, Q)$	test value
	$\mid$	$([n = V]C, P)$	test true
	$\mid$	$([n = V]P, C)$	test false
	$\mid$	$\langle C \rangle$	definition
	$\mid$	C, D	left definition composition
	$\mid$	D, C	right definition composition
	$\mid$	$J = C$	join guarded process

Figure 2.12: Typing Contexts

The type system is defined using typing judgements of the form:

$$\begin{aligned} \Gamma &\vdash C : \tau \\ \Gamma &\vdash P : \tau \\ \Gamma &\vdash D \\ \Gamma &\vdash \mathcal{S} : \tau \end{aligned}$$

where C are extended M-calculus contexts, whose grammar is given in Figure 2.12.

Definition 2.3.1 (Well formed typing environment) *In order to type the input, output and error channels, we only consider in the following typing environments that contain the following bindings:*

$$\begin{aligned} \mathbf{i} &: \forall\rho.(\mathtt{unit} \to \rho) \to \rho \\ \mathbf{o} &: \forall\rho.(\mathtt{unit} \to \rho) \to \rho \\ \mathbf{e} &: \forall\rho.(\mathtt{unit} \to \rho) \to \rho \end{aligned}$$

The type system is defined by the rules in Figure 2.13. They make use of the $Inst$ operator, that takes a type scheme and returns a type where the generalized type variables and multiset variables have been instantiated to types and multisets respectively, and of the predicate set on multisets, which is true of multisets that have at most one occurence of each cell name and no occurrence of any multiset variable.

A few comments on the typing rules are in order. In typing rule NU.DOM, we check that a occurs at most once in Δ, so that a does not occur in the concluding type judgment. In typing rule PASS, we require ρ_1 and ρ_2 to be fresh for the subject reduction of the semantic rule PASS. This implies that in any $\mathtt{pass}_a\ V$,

$$\frac{}{\Gamma \vdash \mathbf{0} : \emptyset}\ [\text{NIL}] \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}}\ [\text{VOID}] \qquad \frac{u : s \in \Gamma \qquad \sigma = Inst(s)}{\Gamma \vdash u : \sigma}\ [\text{NAME}]$$

$$\frac{}{\Gamma \vdash (\,\cdot : \tau) : \tau}\ [\text{PROC.HOLE}] \qquad \frac{\Gamma \vdash a : \mathtt{dom} \qquad \Gamma \vdash r : \sigma \to \Delta}{\Gamma \vdash a.r : \sigma \to \Delta}\ [\text{ADDR}]$$

$$\frac{\Gamma + x : \sigma \vdash P : \tau \qquad x \notin fn(\Gamma)}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\ [\text{FUN}] \qquad \frac{\Gamma \vdash a : \mathtt{dom} \qquad \Gamma \vdash P : \Delta_1 \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash a(P)[Q] : a, \Delta_1, \Delta_2}\ [\text{DOM}]$$

$$\frac{\Gamma \vdash P : \Delta_1 \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash P \mid Q : \Delta_1, \Delta_2}\ [\text{PAR}]$$

$$\frac{\begin{array}{c}\Gamma + r : \forall \widetilde{\alpha}\widetilde{\rho}.\sigma \to \Delta \vdash P : \Delta_1 \\ r \notin fn(\Gamma) \qquad ftv(\forall \widetilde{\alpha}\widetilde{\rho}.\sigma \to \Delta) = fsv(\forall \widetilde{\alpha}\widetilde{\rho}.\sigma \to \Delta) = \emptyset\end{array}}{\Gamma \vdash \nu r : \forall \widetilde{\alpha}\widetilde{\rho}.\sigma \to \Delta.P : \Delta_1}\ [\text{NU.RES}]$$

$$\frac{\Gamma + a : \mathtt{dom} \vdash P : \Delta \qquad a \notin fn(\Gamma) \qquad a \notin (\Delta - a)}{\Gamma \vdash \nu a.P : \Delta - a}\ [\text{NU.DOM}]$$

$$\frac{\begin{array}{c}\Gamma \vdash V : (\mathtt{unit} \to \rho_1) \to (\mathtt{unit} \to \rho_2) \to \Delta \\ \rho_1, \rho_2 \text{ do not occur in } \Gamma \qquad \rho_1, \rho_2 \text{ occur each at most once in } \Delta\end{array}}{\Gamma \vdash \mathtt{pass}_a\ V : \Delta - (a, \rho_1, \rho_2)}\ [\text{PASS}]$$

$$\frac{\Gamma \vdash P : \sigma \to \tau \qquad \Gamma \vdash Q : \sigma' \qquad \sigma' \subseteq \sigma}{\Gamma \vdash PQ : \tau}\ [\text{APP}] \qquad \frac{\Gamma \vdash P : \tau_1 \qquad \Gamma \vdash Q : \tau_2}{\Gamma \vdash [n = V]P, Q : \tau_1 \wedge \tau_2}\ [\text{TEST}]$$

$$\frac{\Gamma \vdash D}{\Gamma \vdash \langle D \rangle : \emptyset}\ [\text{DEF}] \qquad \frac{\Gamma \vdash D_1 \qquad \Gamma \vdash D_2}{\Gamma \vdash D_1, D_2}\ [\text{AND}] \qquad \frac{}{\Gamma \vdash \cdot}\ [\text{DEF.HOLE}] \qquad \frac{}{\Gamma \vdash \bot}\ [\text{DEF.}\bot]$$

$$\frac{\begin{array}{c}(r_i : s_i = \forall \widetilde{\alpha}_i\widetilde{\rho}_i.\widetilde{\sigma_i} \to \Delta_i \in \Gamma)^{i \in [1..n]} \\ \Delta' \subseteq \Delta_1, \ldots, \Delta_n \qquad \Gamma + \widetilde{x_1} : \widetilde{\sigma_1} + \ldots + \widetilde{x_n} : \widetilde{\sigma_n} \vdash P : \Delta' \\ (\widetilde{x_i})^i \cap fn(\Gamma) = \emptyset \qquad \forall i \in [1..n].fv(\widetilde{\sigma_i} \to \Delta_i) \cap fv(\Gamma) \cap (\widetilde{\alpha}_i \cup \widetilde{\rho}_i) = \emptyset \\ \forall i, j \in [1..n]^2.i \neq j \implies fv(\widetilde{\sigma_i} \to \Delta_i) \cap fv(\widetilde{\sigma_j} \to \Delta_j) \cap (\widetilde{\alpha}_i \cup \widetilde{\rho}_i \cup \widetilde{\alpha}_j \cup \widetilde{\rho}_j) = \emptyset\end{array}}{\Gamma \vdash r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} = P}\ [\text{JOIN}]$$

$$\frac{\Gamma \vdash P : \Delta \qquad set(\Delta)}{\Gamma \vdash \epsilon[P] : \Delta}\ [\text{TOP}]$$

$$\frac{\Gamma + r : s \vdash \mathcal{S} : \Delta \qquad r \notin fn(\Gamma) \qquad fsv(s) = ftv(s) = \emptyset}{\Gamma \vdash \nu r : s.\mathcal{S} : \Delta}\ [\text{TOP.NU.RES}]$$

$$\frac{\Gamma + a : \mathtt{dom} \vdash \mathcal{S} : \Delta \qquad a \notin fn(\Gamma)}{\Gamma \vdash \nu a.\mathcal{S} : \Delta - a}\ [\text{TOP.NU.DOM}] \qquad \frac{set(\Delta)}{\Gamma \vdash (\,\cdot : \Delta) : \Delta}\ [\text{TOP.HOLE}]$$

Figure 2.13: Typing rules

V must be an explicit function and not simply a name ($\lambda f.\texttt{pass}_a\ f$ is not accepted by the type system). Note that because of typing rule JOIN, special channels that throw away their result are still well typed (the requirement for the implementation is for the result to be *included* in the exposed type). In rule JOIN, a type for each defined resource is extracted from the type environment, and the guarded process is typed as an immediate instance of that type. Then we check that the generalization is correct, according to the same criteria as for the Join Calculus:

- generalized variables do not occur free in the typing environment;
- generalized variables may not be shared between resources.

Chapter 3

Discussion and examples

This chapter discusses the main features of the M-calculus and presents several examples. In particular, we show how features of distributed process calculi can be simulated in the M-calculus. We also discuss various limitations and possible amendments of the calculus.

3.1 Transparent communications

Compared to the join calculus, which has a fully transparent routing of messages, routing in the M-calculus is not completely transparent. Messages targeting a given reference are not forwarded automatically to the declaration that bears the reference. Instead, messages are routed in a step-by-step fashion from cell to cell. Each step gets the forwarded message closer to its intended destination, that is, the closest enclosing cell containing a definition for the resource for simple messages, or the destination cell for addressed messages (this is the role of the side conditions attached to the message routing rules in Figures 2.9 and 2.10), but the routing process must be explicitly supported by the intervening cell membranes. This endows cell membranes with the ability to filter messages on their way in or out of the cell plasm they control.

It is very easy to define cell membranes that provide for a transparent routing of messages. Cell membranes of the form Fwd below, achieve exactly that:

$$Fwd = \langle \mathbf{i}\ m = m()\ ;\ \mathbf{o}\ m = m() \rangle$$

We remark that in the environment $\mathbf{i} : \forall\rho.(\mathtt{unit} \to \rho) \to \rho, \mathbf{o} : \forall\rho.(\mathtt{unit} \to \rho) \to \rho$ this definition is well typed. Incoming messages, i.e. those targeting a reference or a cell in the cell plasm (rules RED.ADDR.FINAL.PLASM, RED.ADDR.FILTER.IN), and outgoing messages, i.e. those targeting a reference or a cell out of the cell plasm (rules RED.MESS.FILTER.OUT, RED.ADDR.FILTER.OUT), are just released in the cell membrane: they will be further routed according to the message routing rules RED.MESS.PLASM.IN, RED.ADDR.PLASM.IN, RED.MEM.MESS.OUT, RED.PLASM.MESS.OUT, and RED.ADDR.MEM.OUT of the reduction relation.

Notice that this forwarder is completely stateless and holds no specific routing information. It relies entirely on the message routing rules to perform the actual work. Notice also that the forwarder works correctly even in the presence of mobile processes, i.e. even if the targeted cell is moved from cell to cell. Because of the asynchronous nature of the reduction rules in the M-calculus, it is possible, for instance, after having applied rule RED.ADDR.FILTER.IN, and before the message is readied to be moved in the cell plasm by the forwarder, that the side condition of rule RED.ADDR.PLASM.IN no longer holds true because the cell plasm has been changed (see Section 3.3 below for examples). The forwarder is oblivious to this fact and can nevertheless add the message to the cell membrane. In this case, rule RED.ADDR.MEM.OUT

applies and the message can be moved back outside of the cell membrane, as expected. The following example reductions illustrate the situation:

$$
\begin{array}{lcll}
b.r\widetilde{V} \mid a(Fwd)[Q] & \rightarrow & a(\mathbf{i}(\lambda.b.r\widetilde{V}) \mid Fwd)[Q] & \text{by rule RED.ADDR.FILTER.IN; by assumption } b \in \mathsf{df}(Q) \\
& \rightarrow^* & a(\mathbf{i}(\lambda.b.r\widetilde{V}) \mid Fwd)[R] & \text{by assumption, } Q \text{ is changed to } R \text{ and } b \notin \mathsf{df}(R) \\
& \rightarrow^* & a(b.r\widetilde{V} \mid Fwd)[R] & \text{by definition of } Fwd \\
& \rightarrow & b.r\widetilde{V} \mid a(Fwd)[R] & \text{by rule RED.ADDR.MEM.OUT}
\end{array}
$$

The routing situation in presence of mobile processes gets a bit more intricate when one considers the last stages of routing for a message bearing an addressed resource name as above, or when one considers a message bearing a resource name. In the first case, rules RED.ADDR.FINAL.PLASM or RED.ADDR.FINAL.MEM apply. Once in its final target membrane or plasm, a message can no longer be dissociated from its target resource and the re-routing above cannot occur. Because of the filtering implied by rule RED.ADDR.FINAL.PLASM, there is actually a chance that a message targeting an addressed resource in a plasm never reaches its destination if passivation intervenes before the handling of the message on a $\mathbf{i}$ port of the membrane and the triggering of rule RED.MESS.PLASM.IN. If there is no corresponding resource in the cell, rule RED.ADDR.FINAL.MEM applies, and the message will subsequently be handled by the rules of figure 2.9.

In the second case, the message is routed according to the rules in Figure 2.9, which means the message may fail to reach its target resource (if it was not present in the particular cell tree to begin with, or if it has moved along with its encompassing cell) and may end up being handled by default through rule RED.MESS.ERR.

An alternative to the present handling of references to resources would be to consider that resource names have only local significance, i.e. that they must refer to resources located inside the current cell or fail. This would imply the suppression of rules RED.MESS.FILTER.OUT, RED.MEM.MESS.OUT, RED.PLASM.MESS.OUT and the replacement of rule RED.MESS.ERR by the following:

$$
\frac{r \notin \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{a(P \mid r\widetilde{V})[Q] \rightarrow a(P \mid \mathbf{e}(\lambda.r\widetilde{V}))[Q]}\ [\text{RED.MESS.ERR.MEM}]
$$

$$
\frac{r \notin \mathsf{dln}(P) \qquad r \notin \mathsf{dln}(Q)}{a(P)[Q \mid r\widetilde{V}] \rightarrow a(P \mid \mathbf{e}(\lambda.r\widetilde{V}))[Q]}\ [\text{RED.MESS.ERR.PLASM}]
$$

However, it must be pointed out that in this case the notion of dynamic binding is weakened, since resource names no longer reference the closest enclosing resource defining them, but only a resource local to the cell.

3.2 Resource names and dynamic binding

The M-calculus includes two forms of names: *resource names* and *cell names*. Resource names identify definitions as in the join-calculus or resources as in the blue calculus, i.e. process factories that can create new processes upon reception of messages targeting them. Resources can be duplicated, and the same resource name may refer to several different resources. Routing to resource names, however, follows a restricted pattern. Cell names identify cells, i.e. the composition of a membrane process and a plasm process, and are guaranteed by the type system to be unique in evaluation context (see Chapter 4).

Cell names and resource names can be concatenated to form so-called *addressed resource names*. An addressed resource name identifies a resource located in a particular cell. More precisely, an addressed resource name consists of a cell name a concatenated with a resource name r. The name $a.r$ thus refers to a resource of name r, to be found in the membrane or plasm of cell a. The notion of addressed resource name thus reduces the potential ambiguity of non-unique resource names and obtains effects which are similar to definitions in the join-calculus.

Resources provide a form of dynamic binding since the exact interpretation of a resource name is only resolved when routing a message. To illustrate this dynamic binding capability, we discuss below the modeling of libraries in the M-calculus.

Briefly, a library in the M-calculus can be understood as a definition, e.g. a process of the form $\langle \mathsf{l}\ \widetilde{x} = Lib \rangle$, where the name l can used by programs wishing to access the functionality in Lib. In the context of the M-calculus and the presence of cells, one must consider the existence of multiple copies of the same library in different cells, and indeed entertain the possibility of moving a library from one cell to another (e.g. to upgrade a cell membrane or plasm with a new version of an existing library). An important requirement deriving from this is the ability for a program to reference a library by the same name regardless of the particular cell the program currently resides in. Using resource names provides just this capability.

Updating or moving a library is also possible in the M-calculus. Consider updating a library as an example. We are looking for a behavior which would be an analog of the following:

$$a(\langle \mathsf{l}\ \widetilde{x} = L_1 \rangle \mid P(\mathsf{l}))^*[Q] \mid (a.\mathsf{update}\ (\mathsf{l}, \lambda\widetilde{x}.L_2)) \rightarrow^* a(\langle \mathsf{l}\ \widetilde{x} = L_2 \rangle \mid P(\mathsf{l}))^*[Q]$$

where $(\langle \mathsf{l}\ \widetilde{x} = L_i \rangle \mid P(\mathsf{l}))^*$ is an appropriate "variant" of $(\langle \mathsf{l}\ \widetilde{x} = L_i \rangle \mid P(\mathsf{l}))$. We can achieve the desired behavior by defining $(\langle \mathsf{l}\ \widetilde{x} = L \rangle \mid P(\mathsf{l}))^*$ as follows, using the encoding of a reference cell r:

$$\begin{array}{rcl} (\langle \mathsf{l}\ \widetilde{x} = L \rangle \mid P(\mathsf{l}))^* & = & \nu r.((r\ \lambda\widetilde{x}.L) \mid \langle r\ f \mid \mathsf{l}\ \widetilde{y} = f\ \widetilde{y} \mid r\ f \rangle \mid \langle r\ f' \mid (\mathsf{update}\ (x, f)) = A(\mathsf{l}, r, x, f, f') \rangle) \\ A(\mathsf{l}, r, x, f, f') & = & ([\mathsf{l} = x]\ (r\ f),\ (r\ f')) \end{array}$$

where $\widetilde{x}$ and $\widetilde{y}$ have the same size.

Let us suppose that in the previous example, the library resource has type $\mathsf{l} : \widetilde{\sigma} \rightarrow \emptyset$. In this case, the process if well typed with the following types:

$$\begin{array}{rcl} r & : & (\widetilde{\sigma} \rightarrow \emptyset) \rightarrow \emptyset \\ \mathsf{update} & : & (\widetilde{\sigma} \rightarrow \emptyset, \widetilde{\sigma} \rightarrow \emptyset) \rightarrow \emptyset \end{array}$$

We remark that the library cannot be polymorphic, since its implementation is stored in a reference cell that may be updated. More technically, polymorphism if prevented by the sharing of type variables between r and l, and r and update, and the hypothesis of typing rule JOIN.

3.3 Dynamic reconfiguration examples

We investigate in this section various examples of dynamic reconfiguration which can be modelled in the M-calculus.

3.3.1 Creating a new cell

We consider the ability to create a new cell from an existing one. We are looking for the following behavior:

$$a.n\ () \mid a(\langle n\ () = New \rangle \mid P_1)[Q_1] \quad \rightarrow^* \quad b(P_2)[Q_2] \mid a(\langle n\ () = New \rangle \mid P_1)[Q_1]$$

It is trivial to use a declaration to create a new process. The only subtlety here is in making sure the new cell $b(P_2)[Q_2]$ is indeed created *outside* of the original cell. It suffices to define New as:

$$New = \mathtt{pass}_a\ \lambda p\, q.(b(P_2)[Q_2] \mid a(p())[q()])$$

The process New has type b, Δ_2 if Δ_2 is the multiset of cell names active in P_2 and Q_2. To be well typed, a top-level configuration may use n at most once, b, Δ_2 must be a set and should not intersect with other active cells of the configuration.

3.3.2 Adding a process to a cell plasm

We consider the ability to add new processes to a cell plasm. More precisely, we are looking for a definition $\langle \mathsf{add}\ f = \ldots \rangle$ which would provide the following reduction:

$$a.\mathsf{add}\,(\lambda.P) \mid a(\langle \mathsf{add}\ f = Add(a,f)\rangle)[Q] \quad \to^* \quad a(\langle \mathsf{add}\ f = Add(a,f)\rangle)[P \mid Q]$$

Defining $Add(a,f) = \mathtt{pass}_a\ \lambda p\, q.a(p())[f() \mid q()]$ does the job. Indeed, we have the following reductions:

$$\begin{array}{rcl}
a.\mathsf{add}\,(\lambda.P) \mid a(\langle \mathsf{add}\ f = Add(a,f)\rangle)[Q] & \to & a(\mathsf{add}\,(\lambda.P) \mid \langle \mathsf{add}\ f = Add(a,f)\rangle)[Q] \\
& \to & a(\langle \mathsf{add}\ f = Add(a,f)\rangle \mid \mathtt{pass}_a\ \lambda p\, q.a(p())[(\lambda.P)() \mid q()])[Q] \\
& \to & \lambda p\, q.a(p())[(\lambda.P)() \mid q()](\lambda.\langle \mathsf{add}\ f = Add(a,f)\rangle)(\lambda.Q) \\
& \to^* & a(\langle \mathsf{add}\ f = Add(a,f)\rangle)[P \mid Q]
\end{array}$$

Alternatively, one could have defined $Add'(a,f) = (\mathsf{ins}\ f)$, provided Q is of the form $\langle \mathsf{ins}\ f = f()\rangle \mid Q'$. This alternative coding avoids the passivation used in $Add(a,f)$. In both cases the type of add or ins is $\forall\rho.(\mathtt{unit} \to \rho) \to \rho$.

3.3.3 Moving a process to a different cell

Moving a process to a different cell is also straightforward. We have seen above how to add a process to a cell plasm. Moving a process to a different cell thus involves passivating the process and sending it in frozen form to a container that has the add operation. If we take the join calculus and its go primitive as an example, we can think of providing the means to move a whole cell to a different one (i.e. adding it to its plasm). In other terms, we are looking for a construct which behaves like this:

$$a(P)[(\mathsf{go}\ u) \mid Q] \to^* u.\mathsf{add}\,(a(P)[Q]^*)$$

where, intuitively, on carrying out the $(\mathsf{go}\ u)$ instruction, a cell can passivate and send itself (in passivated form $a(P)[Q]^*$) to the definition bearing the reference u (typically, a cell membrane that will add the passivated cell to its plasm). It suffices to define P as:

$$\begin{array}{rcl}
P & = & (Fwd \mid \langle \mathsf{go}\ u = Go(a,u)\rangle) \\
Go(a,u) & = & \mathtt{pass}_a\ \lambda p\, q.u.\mathsf{add}\,(\lambda.a(p())[q()])
\end{array}$$

Surprisingly enough, the process $Go(a,u)$ has type $\emptyset$. This is because this process adds a cell that was passivated, thus the resulting process does not create any new active cell. The resource go has type $\mathtt{dom} \to \emptyset$.

3.3.4 Removing a process from a cell plasm

The go construct above allows for a process (actually a cell) to be passivated and moved to another location (another cell). It can be leveraged to obtain a more "objective" view of move, whereas a process, external to a given cell, can remove a process which appears as a component of the cell plasm. Intuitively, we are looking for a behavior which is analog to the following:

$$(a.\mathsf{move}\ (u,v)) \mid a(P)[Q(u) \mid R] \ \to^* \ (v.\mathsf{add}\ Q^*(u)) \mid a(P)[R]$$

where $Q(u)$ denotes a "component" designated by name u, $Q^*(u)$ denotes a "passivated" form of $Q(u)$, and where v is assumed to be a cell outside of $a(P)[R]$.

A named component in the M-calculus can be readily represented by a cell. We can define $Q(b)$ by: $Q(b) = b(C_b)[Q]$, where

$$C_b = (Fwd \mid \langle \mathsf{go}\ v = Go(b, v)\rangle)$$

We obtain the required behavior by defining P as:

$$P = (Fwd \mid \langle \mathsf{move}\ (x, y) = (x.\mathsf{go}\ y)\rangle)$$

We have the following reductions:

$$\begin{array}{rcl}((a.\mathsf{move}\ (b, v)) \mid a(P)[Q(b) \mid R]) & \rightarrow & a((\mathsf{move}\ (b, v)) \mid P)[Q(b) \mid R] \\ & \rightarrow^* & a(P)[(b.\mathsf{go}\ v) \mid Q(b) \mid R] \\ & \rightarrow^* & a(P)[v.\mathsf{add}\ (\lambda.b((\lambda.C_b)())[(\lambda.Q)()] \mid R] \\ & \rightarrow^* & a(v.\mathsf{add}\ (\lambda.b((\lambda.C_b)())[(\lambda.Q)()]) \mid P)[R] \\ & \rightarrow & (v.\mathsf{add}\ (\lambda.b((\lambda.C_b)())[(\lambda.Q)()]) \mid a(P)[R])\end{array}$$

where we can interpret $Q^*(b)$ as $(\lambda.b((\lambda.C_b)())[(\lambda.Q)()])$, i.e. the frozen form of the cell $b(C_b)[Q]$. In this example, process P above can be understood as a *component container* in the sense of [20, 24]. As in the previous example, we may type move with the type $\mathtt{dom}, \mathtt{dom} \rightarrow \emptyset$

3.3.5 Controllable components

The notion of movable component which we obtained in the previous section can be generalized to achieve various forms of control on individual processes in the M-calculus. As a first example, we define an interruptible component $Q^i(b)$ of name b and behavior Q as:

$$\begin{array}{rcl}Q^i(b) & = & \nu r\ \mathsf{on}.b(Fwdr(\mathsf{on}) \mid \langle \mathsf{suspend} \mid \mathsf{on} = Suspend_b; \mathsf{resume} \mid r\ x = Add(b, x) \mid \mathsf{on}\rangle \mid \mathsf{on})[Q] \\ Fwdr(\mathsf{on}) & = & \langle \mathsf{on} \mid \mathbf{i}\ m = m() \mid \mathsf{on}\ ;\ \mathsf{on} \mid \mathbf{o}\ m = m() \mid \mathsf{on}\rangle \\ Suspend_b & = & \mathtt{pass}_b\ \ \lambda p\ q.b(p\ () \mid (r\ q))[0]\end{array}$$

In this example, the process Q is frozen in its current state upon receipt of the message suspend by its controller. It resumes its operation upon receipt of the resume message by its controller (the on message is used to ensure the atomic semantics of suspend and resume operations). The storage message r may be given the type $\forall\rho.(\mathtt{unit} \rightarrow \rho) \rightarrow \rho$. The $Suspend_b$ process has type $\emptyset$ since it recreates the cells that is passivated with the plasm stored in the storage message, ready to be reactivated by the resume message.

Using a construct similar as that defined in section 3.2 above, one can likewise define an evolvable component $Q^e(b)$ of name b and behavior Q in which the controller behavior L is an updatable library. A more radical form would consist in a cell membrane that upgrades its behavior entirely upon receipt of an update message (carrying the new controller behavior in the form of an abstraction):

$$\begin{array}{rcl}Q^e(b) & = & b(P \mid \langle \mathsf{update}\ f = Update_b(f)\rangle)[Q] \\ Update_b(f) & = & \mathtt{pass}_b\ \ \lambda p\ q.b(f())[q()]\end{array}$$

In this example, the resource update may be typed with type $\forall\rho.(\mathtt{unit} \rightarrow \rho) \rightarrow \rho$. This type however does not reflect that the previous controller P disappears when it is updated. If this controller contains an active cell a, and the new controller installed by the update resource also contains an active cell a, the configuration will not be well typed although it is correct. Much more complex types would be needed to accept these processes.

The different forms of components we have defined above can of course be combined. We could, for instance, envisage named, movable, interruptible, evolvable components $Q^{mie}(b)$.

The important point here is that the behavior of individual processes in the M-calculus can thus be externally controlled and managed by "component containers".

3.4 Simulating distributed process calculi

In this section, we present, by means of examples, evidence that the M-calculus adequately captures crucial features of recent distributed process calculi. We consider the distributed join calculus [10, 13], and the π_{1l} calculus [1].

3.4.1 Simulating the π_{1l}-calculus

The π_{1l}-calculus has three distinguishing features: named localities, where processes reside, and which are organized as a flat parallel composition; a spawn primitive, which allows a process to move from one locality to another; silent failures of localities with a simple faithful failure detector which takes the form of a ping primitive.

Simulating the π_{1l}-calculus with the M-calculus is straightforward. A π_{1l}-calculus locality $a[\cdot]$ is modelled as a cell of the form $a(PP(a))[\cdot]$ where $PP(a)$ is defined as follows:

$$\begin{aligned} PP(a) &= \nu \mathsf{on}\ \mathsf{off}.(\langle \mathsf{on} \mid \mathbf{i}\, m = (\mathsf{on} \mid m()) \ ;\ \mathsf{on} \mid \mathbf{o}\, m = (\mathsf{on} \mid m())\rangle \\ &\quad \mid \langle \mathsf{on} \mid \mathsf{add}\ \ f = Augment(a, f) \\ &\quad\ \ \mathsf{on} \mid \mathsf{stop} = \mathsf{off} \\ &\quad\ \ \mathsf{on} \mid \mathsf{ping}\ (y, n) = \mathsf{on} \mid y() \\ &\quad\ \ \mathsf{off} \mid \mathsf{ping}\ (y, n) = \mathsf{off} \mid n()\rangle) \\ Augment(a, f) &= \mathtt{pass}_a\ \lambda p\ q.a(\mathsf{on} \mid p())[q() \mid f()] \end{aligned}$$

The $\mathsf{spawn}(a, P)$ construct of the π_{1l}-calculus is a simple move of a process P to a cell (locality) named a. It can be simply encoded as $(a.\mathsf{add}\ \ \lambda.P)$. The $\mathsf{stop}(a)$ and $\mathsf{ping}(a, y, n)$ constructs of the π_{1l}-calculus can be likewise encoded as $a.\mathsf{stop}$ and $a.\mathsf{ping}\ (y, n)$, respectively.

As in previous examples, the resource add has type $\forall\rho.(\mathtt{unit} \to \rho) \to \rho$, and ping may be given type $(\mathtt{unit} \to \emptyset, \mathtt{unit} \to \emptyset) \to \emptyset$.

3.4.2 Simulating the distributed join calculus

Simulating the distributed join calculus with the M-calculus is straightforward. A join calculus locality $a[\,\cdot\,]$ can be modelled by a cell of the form $a(PJ(a))[\,\cdot\,]$, where $PJ(a)$ is defined as follows:

$$\begin{aligned} PJ(a) &= (Fwd \mid \langle \mathsf{add}\ f = Add(a, f)\rangle \mid \langle \mathsf{go}\ b = Send(a, b)\rangle) \\ Add(a, f) &= \mathtt{pass}_a\ \lambda p\ q.a(p())[q() \mid f()] \\ Send(a, b) &= \mathtt{pass}_a\ \lambda p\, q.(b.\mathsf{add}\ \lambda.a(p())[q()]) \end{aligned}$$

The $\mathsf{go}\langle b\rangle$ construct of the distributed join calculus can be encoded as $(\mathsf{go}\ b)$, join calculus definitions as M-calculus definitions, and join calculus channel names as addressed resource names of the form $a.r$, where a is the location where r is defined. As before, add has type $\forall\rho.(\mathtt{unit} \to \rho) \to \rho$ and go has type $\mathtt{dom} \to \emptyset$.

Encoding the fail-stop behavior of the distributed join calculus localities and their associated faithful failure detectors can also be done in the M-calculus (see [5]), however the encoding is non trivial because each cell simulating a join calculus locality must keep track of its subcells, multicast stop failure messages to them, and forward them ping failure detection messages. The complexity in the encoding is due to the limited form of message reification that is available in the M-calculus. A much simpler encoding is discussed in the section below.

J	::=		join pattern
		$r\widetilde{p}$	message with pattern
		$J \mid J$	join
p	::=		receipt pattern
		x	variable
		$-$	any match
		(p, p)	pair

Figure 3.1: Syntax modifications for the extended M-calculus

3.5 Reflective features

The M-calculus contains two reflective features: the reification of messages, provided by interception rules (RED.MESS.FILTER.OUT, RED.ADDR.FILTER.OUT, RED.ADDR.FINAL.PLASM, and RED.ADD.FILTER.IN), and the passivation of cells, provided by rule RED.PASSIV. In both cases, the use of reflection is kept at a minimum. Using the pass construct, it is only possible to freeze a cell membrane and a cell plasm in their current state, and to unfreeze them later on (if at all), possibly in a different cell. The current calculus does not provide access to the details of a given configuration (cell membrane or cell plasm). The interception rules are limited in the same fashion: the current calculus does not allow to inspect the details of intercepted messages, only to delete them or free them in a different context.

This limited forms of reflection are unsatisfactory, if cells in the M-calculus are intended to capture directly and accurately existing forms of cells such as, e.g. EJB component containers [24], or faulty machine nodes.

Another limitation of the reflective capabilities of the current calculus has to do with the observation capabilities of a cell membrane. By encapsulating a process within a component as in section 3.3.5 it is possible to observe incoming and outgoing messages, but internal communications, i.e. message exchanges taking place between the parallel components of the enclosed process, as well as the enclosed process state (especially as manifested by waiting messages) remain invisible to the cell membrane. It thus appears impossible to program, in the current calculus, general observers as defined in [23].

Without resorting to a fully reflective calculus, it is possible to extend the current calculus with limited reflective features that should still be typable with a relatively simple type system. The extension we can consider comprise a full reification of messages targeting the special ports $\mathbf{i}$, $\mathbf{o}$ and $\mathbf{e}$. The syntax of the calculus can be modified as Figure 3.1, where receipt patterns are constrained to be linear, i.e. in a pattern $p = (x_1, (x_2, \ldots, (x_{k-1}, x_k)\ldots)$ variables x_j are all distinct.

Reduction rules of the extended calculus are those of the M-calculus, except for rule RED.RES which is replaced by rule RED.RES.PATTERN below:

$$\frac{\langle D\rangle = \langle D_0 \;;\; r_1\widetilde{p_1} \mid \ldots \mid r_n\widetilde{p_n} = P\rangle \qquad \widetilde{p_j}\theta = \widetilde{V_j},\ j \in \{1,\ldots,n\}}{\langle D\rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} \to \langle D\rangle \mid P\theta} \ [\text{RED.RES.PATTERN}]$$

and for routing rules RED.MESS.FILTER.OUT, RED.MESS.ERR, RED.ADDR.FINAL.PLASM, RED.ADDR.FILTER.IN, RED.ADDR.FILTER.OUT, where occurrences of the form $\mathsf{w}(\lambda.sV_1 \ldots V_k)$ (with $\mathsf{w} \in \{\mathbf{i}, \mathbf{o}, \mathbf{e}\}$), are replaced by $\mathsf{w}(s, (V_1, (V_2, (\ldots, V_k)\ldots)))$.

With these modifications to the calculus, it is now possible for a membrane to make its behavior dependent on the analysis of messages targeting definitions in the cell plasm it controls. As an illustration, we

can provide a straightforward interpretation in the extended M-calculus of the distributed join calculus with failures.

A locality $a[\,\cdot\,]$ in the distributed join calculus with failure can be modelled by a cell of the form $a(PJF(a))[\,\cdot\,]$, where $PJF(a)$ is defined as follows:

$$
\begin{array}{rcl}
PJF(a) & = & \nu\mathsf{on}.(\langle \mathsf{on} \mid \mathbf{i}\, m = (\mathsf{on} \mid m)\rangle \\
& & \quad \mid \langle \mathsf{on} \mid \mathbf{o}\, m = (\mathsf{on} \mid m)\rangle \\
& & \quad \mid \langle \mathsf{on} \mid \mathsf{ins}\ f = Insert(a, f)\rangle \\
& & \quad \mid \langle \mathsf{on} \mid \mathsf{go}\ b = Send(a, b)\rangle \\
& & \quad \mid \langle \mathsf{on} \mid \mathsf{halt} = Halt(a)\rangle \\
& & \quad \mid \langle \mathsf{on} \mid \mathsf{ping}\ (y, n) = (\mathsf{on} \mid y\,())\rangle) \\
Insert(a, f) & = & \mathtt{pass}_a\ \lambda p\ q.a(\mathsf{on} \mid p())[q() \mid f()] \\
Send(a, b) & = & \mathtt{pass}_a\ \lambda p\, q.(b.\mathsf{ins}\ \lambda.a(\mathsf{on} \mid p())[q()]) \\
Halt(a) & = & \mathtt{pass}_a\ \lambda p\ q.\ a(\langle \mathsf{ping}\ (y, n) = n\,()\rangle \mid \langle \mathbf{i}\ (\mathsf{b.ping}, (y, n)) = n\,()\rangle \mid \langle \mathbf{o} - = 0\rangle)[q()]
\end{array}
$$

A failed locality is thus modelled as a cell which only responds (negatively) to ping failure detection messages (notice that even though the cell plasm is not frozen, it is completely isolated from the outside world). Since the failure of a locality in the distributed join calculus with failure implies the failure of the sublocalities, a failed cell also responds to ping messages addressed to its subcells. Such messages necessarily appear on its $\mathbf{i}$ port by rule RED.ADDR.FILTER.IN, where they can be examined to obtain the return address of the failure detection message.

A type system for such an extended calculus also requires reflective features, to be able to type tuples of variable size, for instance. We are currently working on such an extension.

Chapter 4

Subject reduction

This chapter is dedicated to the proof of the subject reduction theorem for the type system of the M-calculus. An immediate corollary of the theorem is the preservation of the property of unicity of active cell names in well-typed top-level configurations.

4.1 Preliminary lemmas

Proposition 4.1.1 (Properties of type inclusion) *1. The "$\subseteq$" relation is a partial order on types.*

2. *Let θ be a substitution from type variables to types and from multiset variables to multiset. If $\tau_1 \subseteq \tau_2$, then $\tau_1\theta \subseteq \tau_2\theta$.*
3. *If $\tau_1' \subseteq \tau_1$, $\tau_2' \subseteq \tau_2$, and if $\tau_1 \wedge \tau_2$ is defined, then we have $\tau_1' \wedge \tau_2' \subseteq \tau_1 \wedge \tau_2$.*
4. *Let τ_1 be a type. For any τ_2 such that $\tau_1 \wedge \tau_2$ is defined, we have $\tau_1 \subseteq (\tau_1 \wedge \tau_2)$.*

Proof: We note first that if $\tau \subseteq \tau'$, then the size of τ is smaller than the size of τ', where the size of a type τ is defined by induction as follows : $size(\Delta) = card(\Delta)$, $size(\texttt{unit}) = size(\texttt{dom}) = size(\alpha) = 0$, $size(\sigma \to \tau) = size(\sigma) + size(\tau)$. This is immediate by induction on the type inclusion relation.

To prove property (1), we first show that "$\subseteq$" is reflexive, by induction on the type structure. This is the case for multisets. This is immediately the case for `unit`, `dom` and type variables. This is the case by induction for tuples and functions.

We now prove that "$\subseteq$" is antisymetric. Let τ_1 and τ_2 such that $\tau_1 \subseteq \tau_2$ and $\tau_2 \subseteq \tau_1$. We prove that $\tau_1 = \tau_2$ by induction on the sum of the size of the types. The base cases (`dom`, `unit`, and type variables) are immediate. The result is immediate by induction for tuples, functions, and multisets.

We now prove that "$\subseteq$" is transitive, by induction on the sum of the size of the three types. This is immediate for `dom`, `unit`, and type variables, as one rule may only apply. This is also immediate by induction for tuples and functions as one rule may also only apply. This is true for multisets.

We now prove property (2). This property holds for multiset inclusion, and is immediate by induction for the other types.

We now prove property (3) by induction on the sum of the size of τ_1 and τ_2. The property immediately holds for multisets (considering the number of occurrences of each name). The `unit`, `dom` and type variable cases are immediate. The tuple and function case are immediate by induction.

We now prove property (4). We proceed by induction on the sum of the size of τ_1 and τ_2. The property is immediate for multisets. The `unit`, `dom` and type variable cases are immediate. The tuple and function cases are immediate by induction. □

Lemma 4.1.2 *If $\Gamma \vdash \mathcal{S} : \Delta$, then $fsv(\mathcal{S}) = ftv(\mathcal{S}) = \emptyset$. The same holds for all syntactical classes (except contexts).*

Proof: Immediate by induction on the typing derivation, as the only rules that deal with type and multiset variables in processes checks that they are not free (NU.RES and TOP.NU.RES). □

Lemma 4.1.3 *If $\Gamma \vdash \mathcal{S} : \Delta$ then $set(\Delta)$.*

Proof: Immediate by induction on the type derivation for the top-level configuration (it suffices to inspect rules TOP, TOP.NU.RES and TOP.NU.DOM). □

Lemma 4.1.4 *Let $\Gamma \vdash P : \tau$ be a type judgement and n' be a name or variable that does not occur in the typing derivation. We have $\Gamma\{^{n'}/_{n}\} \vdash P\{^{n'}/_{n}\} : \tau\{^{n'}/_{n}\}$. The same holds for typing top-level configurations and definitions.*

Proof: By induction on the typing derivation.

Most cases are immediate by induction, as n' does not occur in the types. We detail the harder cases.

TOP We still have $set(\Delta\{^{n'}/_{n}\})$ and we can conclude by induction.

TOP.NU.DOM If n is a, then the substitution does nothing (since $a \notin fn(\Gamma)$ and a is not in $\Delta - a$ (because $set(\Delta)$ by lemma 4.1.3)). Otherwise, immediate by induction (we recall that n' cannot be a).

TOP.NU.RES If n is r, then the substitution does nothing, since r does not occur in Γ and Δ contains only cell names. Otherwise the result is immediate by induction, since the condition on multiset and type variables is not modified.

TOP.HOLE Immediate since we still have $set(\Delta\{^{n'}/_{n}\})$.

NAME As neither names nor variables may be generalized, the result is immediate by induction.

FUN If n is x, then the substitution does nothing (variables do not occur in types). Otherwise it is immediate by induction, since n' is fresh (thus different from x).

NU.RES If the substituted name is r, then as case SOUP.NU.RES the substitution does nothing as $r \notin fn(\Gamma)$ and $r \notin fn(\tau)$ (there are only cell names in types, no resource names). Otherwise it is immediate by induction.

NU.DOM As for case NEW.RES, if the name substituted is a, the substitution does nothing as the name occurs nowhere in the conclusion. Otherwise, it is immediate by induction.

PASS The result holds because neither n nor n' may be ρ_1 nor ρ_2. If n is a, the result holds because $(\mathtt{pass}_a\ V)\{^{a'}/_{a}\} = \mathtt{pass}_{a'}\ V\{^{a'}/_{a}\}$.

APP Immediate by induction, since type inclusion is preserved by renaming.

TEST Immediate, as the "$\wedge$" operator on types commutes with the substitution if it is to fresh names.

JOIN As for case FUN, immediate by induction, as names are distinct from type and multiset variables.

□

In the following lemma, we write $\nu n.(\dots)$ for $\nu r : s.(\dots)$ or $\nu a.(\dots)$. We recall that resource names do not occur in types, thus are not bound by the restriction.

Lemma 4.1.5 (α-conversion of names) *If $\Gamma \vdash \nu n.\mathcal{S} : \Delta$ (resp. $\Gamma \vdash \nu n.P : \tau$), for any fresh name n' we have $\Gamma \vdash \nu n'.(\mathcal{S}\{^{n'}/_{n}\}) : \Delta$ (resp. $\Gamma \vdash \nu n'.(P\{^{n'}/_{n}\}) : \tau$).*

If $\Gamma \vdash \lambda x.P : \tau$, for any fresh variable y we have $\Gamma \vdash \lambda y.P\{^{y}/_{x}\} : \tau$.

If $\Gamma \vdash r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} = P$, for any fresh variables $(\widetilde{y_i})^{i\in[1..n]}$ (with same size tuples than the $\widetilde{x_i}$), we have $\Gamma \vdash r_1\widetilde{y_1} \mid \ldots \mid y_n\widetilde{x_n} = P\{^{\widetilde{y_i}}/_{\widetilde{x_i}}\}$

Proof: Immediate application of lemma 4.1.4 on the hypothesis of rules TOP.NU.DOM, TOP.NU.RES, NU.RES, NU.DOM, FUN, or JOIN remarking neither n nor x can occur in the types (if it is a cell name, it is because $\Delta - a = \Delta\{^{a'}/_{a}\} - a'$ if a occurs at most once in Δ, and if a' is fresh; if it is a resource name or a variable, it cannot occur in a type). □

Lemma 4.1.6 (Typing and contexts) *Let $C(\cdot : \Delta)$ be a top-level configuration context (resp. $C(\cdot : \tau)$ a process context, resp. $C(\cdot)$ a definition context) and $\mathcal{S}$ a top-level configuration (resp. P a process, resp. D a definition) such that $\Gamma \vdash C(\cdot : \Delta) : \Delta_1$ (resp. $\Gamma \vdash C(\cdot : \tau) : \Delta_1$, resp. $\Gamma \vdash C(\cdot) : \Delta_1$) and $\Gamma' \vdash \mathcal{S} : \Delta'$ (resp. $\Gamma' \vdash P : \tau'$, resp. $\Gamma' \vdash D$) with $\Delta' \subseteq \Delta$ (resp. $\tau' \subseteq \tau$) where Γ' is the typing environment when typing the hole.*

Then, $\Gamma \vdash C(\mathcal{S}) : \Delta_1'$ (resp. $\Gamma \vdash C(P) : \Delta_1'$, resp $\Gamma \vdash C(D) : \Delta_1'$) with $\Delta_1' \subseteq \Delta_1$.

The same property holds if the resulting term is a process or a definition, and this property holds with no type inclusion if the top-level configuration, process, or definition that is plugged in has the same type than the hole.

Proof: We proceed by induction on the context size for any term plugged into the context satisfying the condition. The property for identical type (no type inclusion) is immediate.

$\nu a.C$ Immediate by induction using rule TOP.NU.DOM, or by using rule NU.DOM, since in this case the condition on Δ is necessarily satisfied.

$\nu r : s.C$ Immediate by induction using rule TOP.NU.RES or rule NU.RES.

$\epsilon[C]$ Immediate by induction using rule TOP.

$(\cdot : \Delta)$ Since $\Delta' \subseteq \Delta$, we necessarily have $set(\Delta')$ and we conclude by rule TOP.HOLE. In this case the typing environment Γ and Γ' are necessarily the same.

$(\cdot : \tau)$ Immediate, with $\Gamma' = \Gamma$.

$\lambda x.C$ Immediate by induction using typing rule FUN, since $\tau' \subseteq \tau \implies \sigma \to \tau' \subseteq \sigma \to \tau$.

$a(C)[Q]$ **and** $a(P)[C]$ Immediate by induction, applying typing rule DOM.

$C \mid Q$ **and** $P \mid C$ Immediate by induction, using typing rule PAR.

$\mathtt{pass}_a\ C$ Immediate by induction using typing rule PASS, since we necessarily have $\Delta' \subseteq \Delta$, thus the condition on ρ_1 and ρ_2 is still satisfied.

CQ **and** PC Immediate by induction using typing rule APP, relying on the transitivity of type inclusion for the second case (proposition 4.1.1(1)).

$[n = C]P, Q$ **and** $[n = V]C, Q$ **and** $[n = V]P, C$ Immediate by induction using typing rule TEST, since the "$\wedge$" operator preserves type inclusion (proposition 4.1.1(3)).

$\langle C \rangle$ Immediate by induction using typing rule DEF.

C, D' **and** D, C Immediate by induction using typing rule AND.

$(\cdot)$ Immediate by rule DEF.HOLE, with $\Gamma' = \Gamma$.

$J = C$ Immediate by induction using typing rule JOIN.

□

In the following, we use lemmas 4.1.5 and 4.1.6 to work up to α-conversion of names bound by a ν, a λ, or received names of a join pattern.

Lemma 4.1.7 (Instantiation of multiset and type variables) *Let $\Gamma \vdash P : \tau$ be a type judgement, and θ a substitution from type variables to types and from multiset variables to multisets such that $\theta\theta = \theta$. We have $\Gamma\theta \vdash P : \tau\theta$. The same holds for top-level configurations.*

Proof: We first remark that no type nor multiset variable may occur free in a process of a definition (lemma 4.1.2). We proceed by induction on the typing derivation, for any substitution.

NIL,VOID Immediate.

NAME We suppose the generalized variables of s are all different from the variables occurring in the domain and range of θ (renaming them if necessary). Let θ' be the instantiation substitution (we have $\sigma = s\theta'$). We have $u : s\theta \in \Gamma\theta$. Moreover, the substitution $\theta'\theta$ is a correct instantiation. We now show that $s\theta\theta'\theta = s\theta'\theta$. We first consider α, a generalized variable of s. By hypothesis, we have $\alpha\theta = \alpha$, and the resulting type or multiset is the same. We now consider that α is a variable that is in the domain of θ. By hypothesis, we have $\alpha\theta\theta' = \alpha\theta$, and $\alpha\theta' = \alpha$, thus $\alpha\theta\theta'\theta = \alpha\theta\theta = \alpha\theta = \alpha\theta'\theta$.

ADDR Immediate by induction.

FUN Immediate by induction, since process variables do not occur in types.

DOM,PAR Immediate by induction.

NU.RES, TOP.NU.RES Immediate by induction, since resource names do not occur in types, and there are no free type or multiset variable in the binding that is added to Γ.

NU.DOM, TOP.NU.DOM Immediate by induction, after applying some α-conversion if a is in the range of θ.

PASS If ρ_1 or ρ_2 occurs either in the domain or the range of θ, we first rename them, using the induction hypothesis and the substitution $\{^{\rho_1', \rho_2'}/_{\rho_1, \rho_2}\}$ where ρ_1' and ρ_2' occur neither in Γ nor in the domain and range of θ. Thus we have a typing:

$$\Gamma \vdash V : (\mathtt{unit} \to \rho_1') \to (\mathtt{unit} \to \rho_2') \to \Delta_3\{^{\rho_1', \rho_2'}/_{\rho_1, \rho_2}\}$$

Since the other condition are still satisfied, we apply rule PASS to yield:

$$\Gamma \vdash \mathtt{pass}_a\ V : \Delta_3 - (a, \rho_1, \rho_2)$$

(we have $\Delta_3\{^{\rho_1', \rho_2'}/_{\rho_1, \rho_2}\} - (a, \rho_1', \rho_2') = \Delta_3 - (a, \rho_1, \rho_2)$ as ρ_1 and ρ_2 occur at most once in Δ_3).

We may now apply the induction hypothesis with the substitution θ, and the result is immediate by induction.

APP,TEST Immediate by induction (type substitution preserves type inclusion).

DEF,AND Immediate by induction.

DEF.$\bot$ Immediate.

JOIN Immediate by induction after renaming the generalized variables in the resource bindings such that they are different from the domain and variables in the range of θ. Thus the generalization conditions still hold true. The condition on the $\widetilde{x_i}$ still holds as process variables are not affected by the substitution.

TOP Immediate.

□

Lemma 4.1.8 (Type environment extension) *If* $\Gamma \vdash P : \tau$ *and* $u \notin dom(\Gamma)$, *then* $\Gamma + u : \sigma \vdash P : \tau$.

Proof: By induction on the typing derivation.
Most cases are immediate by induction. We detail the other cases.

FUN If x is u, we first α-rename x to a fresh variable (x cannot occur in σ).

NU.RES If r is u, we first α-rename r to a fresh resource name (r cannot occur in σ).

NU.DOM If a is u or occurs in σ, we first α-rename a to a fresh cell name.

PASS If either ρ_1 or ρ_2 occur in σ, we rename them using lemma 4.1.7. As in the case for PASS in the proof of this lemma, the resulting judgement is the same. We may then apply the induction hypothesis, and the result is immediate.

JOIN Immediate by induction, after α-renaming the $\widetilde{x_i}$ if necessary, as well as the generalized variables so that they do not clash with $u : \sigma$ for the generalization condition.

□

Lemma 4.1.9 (Type environment strengthening) *If* $\Gamma + a : dom \vdash P : \tau$ *and* $a \notin fn(P)$, *then* $\Gamma \setminus a \vdash P : \tau \setminus a$.
If $\Gamma + r : s \vdash P : \tau$ *and* $r \notin fn(\Gamma) \cup fn(P)$, *then* $\Gamma \vdash P : \tau$.
The same property holds for top-level configurations.

Proof: By induction on the type derivation. We prove each property in parallel, for top-level configurations, processes and definitions, but detail the first property.

TOP Immediate by induction, since $set(\Delta) \implies set(\Delta \setminus a)$.

TOP.NU.DOM Immediate by induction as the new name cannot be a (a is a free name in $\Gamma + a : \mathtt{dom}$).

TOP.NEW.RES Immediate by induction as the new name cannot be r, and as a cannot occur in the binding added to Γ (otherwise it would be free in the term).

NIL,VOID Immediate.

NAME Immediate if a does not occur in σ, as it cannot occur free in s, thus $s = s \setminus a$. Otherwise, a may occur in σ because it occurs free in s, or because a generalized variable of s was instantiated to a type containing a. As concerns the first kind of occurrences, they do not occur in $s \setminus a$ and also do not occur in $\sigma \setminus a$. As concerns the second kind of occurrences, we replace the instantiation θ with the instantiation $\theta \setminus a$ that associates $\tau \setminus a$ to α if τ is associated to α in θ, and that associates $\Delta \setminus a$ to ρ if Δ is associated to ρ in θ. The resulting type is $\sigma \setminus a$.

ADDR Immediate by induction, as the target cell is not a (it would be free in the term otherwise).

FUN Immediate by induction, considering the binding $x : \sigma \setminus a$ added to Γ.

DOM Immediate by induction, since a cannot be the name of the cell.

PAR Immediate by induction.

NU.RES Immediate by induction as a may not appear in the binding added to Γ.

NU.DOM Immediate by induction as a may not be the new name, since a initially occurs in Γ.

PASS Immediate by induction, as a cannot be the name that is passivated.

APP,TEST Immediate by induction (removing cell names preserve type inclusion).

DEF Immediate by induction.

AND Immediate by induction.

DEF.$\bot$ Immediate.

JOIN Immediate by induction as a may not be a generalized variable, and r may not be one of the resources in the join pattern (the inclusion of Δ' is preserved).

□

4.2 Subject reduction and progress theorems

Lemma 4.2.1 (Subject reduction for $\equiv$) *If $\Gamma \vdash \mathcal{S} : \Delta$ and $\mathcal{S} \equiv \mathcal{S}'$ then $\Gamma \vdash \mathcal{S}' : \Delta$. The same property is true for processes.*

Proof: By induction on the derivation of the structural equivalence (reflexivity and transitivity are immediate, symmetry is dealt with in each case).

STRUCT.NU.PAR We consider the equivalence: $(\nu n.P) \mid Q \equiv \nu n.(P \mid Q)$

There are several cases, depending on whether n is a resource or a cell name, and whether we consider the scope extrusion or the scope intrusion. In any case, we have $n \notin fn(Q)$.

We first consider scope extrusion.

If n is a resource, we first have the typing:

$$\frac{\dfrac{\Gamma + r : s \vdash P : \Delta_1 \qquad r \notin fn(\Gamma)}{\Gamma \vdash \nu r : s.P : \Delta_1}\,[\text{NU.RES}] \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash (\nu r : s.P) \mid Q : \Delta_1, \Delta_2}\,[\text{PAR}]$$

with some additional conditions on s that we do not detail. Since we have $r \notin fn(\Gamma)$, we may use lemma 4.1.8, to yield the judgement (where some names or variable bound in Q might have been renamed):

$$\Gamma + r : s \vdash Q : \Delta_2$$

We conclude by rules PAR and NU.RES.

If n is a cell name a, we suppose in the following derivation that a does not occur in Δ_2, otherwise we α-rename a in $\nu a.P$. We have the typing:

$$[\text{NU.DOM}]\ \frac{\dfrac{\Gamma + a : \mathtt{dom} \vdash P : \Delta_1 \qquad a \notin fn(\Gamma) \qquad a \notin (\Delta_1 - a)}{\Gamma \vdash \nu a.P : \Delta_1 - a} \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash (\nu a.P) \mid Q : (\Delta_1 - a), \Delta_2}\ [\text{PAR}]$$

As in the previous case, we use the hypothesis $a \notin fn(\Gamma)$ to apply lemma 4.1.8 to yield the judgement:

$$\Gamma + a : \mathtt{dom} \vdash Q : \Delta_2$$

Since $a \notin \Delta_2$, we have $a \notin ((\Delta_1, \Delta_2) - a) = (\Delta_1 - a), \Delta_2$. We conclude by rules PAR and NU.DOM.

We now consider scope intrusion. In the case of resources, we initially have the typing:

$$\frac{\dfrac{\Gamma + r : s \vdash P : \Delta_1 \qquad \Gamma + r : s \vdash Q : \Delta_2}{\Gamma + r : s \vdash P \mid Q : \Delta_1, \Delta_2}\ [\text{PAR}] \qquad r \notin fn(\Gamma)}{\Gamma \vdash \nu r : s.P \mid Q : \Delta_1, \Delta_2}\ [\text{NU.RES}]$$

Since $r \notin fn(\Gamma)$ and $r \notin fn(Q)$, we may apply lemma 4.1.9 to yield:

$$\Gamma \vdash Q : \Delta_2$$

We conclude by rule NU.RES on the typing judgement for P and then by rule PAR.

We now suppose that n is a cell name a. We have the typing:

$$[\text{PAR}]\ \frac{\Gamma + a : \mathtt{dom} \vdash P : \Delta_1 \qquad \Gamma + a : \mathtt{dom} \vdash Q : \Delta_2}{\begin{array}{c}\Gamma + a : \mathtt{dom} \vdash P \mid Q : \Delta_1, \Delta_2 \\ \vdots \\ \dfrac{a \notin fn(\Gamma) \qquad a \notin ((\Delta_1, \Delta_2) - a)}{\Gamma \vdash \nu a.P \mid Q : (\Delta_1, \Delta_2) - a}\ [\text{NU.DOM}]\end{array}}$$

Since $a \notin fn(Q)$, we may apply lemma 4.1.9 to yield the judgement:

$$\Gamma \setminus a \vdash Q : \Delta_2 \setminus a$$

We first remark that since $a \notin fn(\Gamma)$, we have $\Gamma \setminus a = \Gamma$.

Since by hypothesis we have $a \notin ((\Delta_1, \Delta_2) - a)$, necessarily a occurs at most once in Δ_1, Δ_2. If a does not occur in Δ_2, then we have $\Delta_2 \setminus a = \Delta_2$, and $(\Delta_1, \Delta_2) - a = (\Delta_1 - a), (\Delta_2 \setminus a)$. Otherwise, a occurs once in Δ_2 and does not occur in Δ_1, and we have $(\Delta_1, \Delta_2) - a = (\Delta_1 - a), (\Delta_2 - a) = (\Delta_1 - a), (\Delta_2 \setminus a)$.

In all cases, the resulting type after applying NU.DOM on the judgement for P and PAR with the judgement for Q is the same than the initial type.

STRUCT.NU.TOP We consider the equivalence: $\epsilon[\nu n.P] \equiv \nu n.\epsilon[P]$.

Again we distinguish between resource names and cell names, scope extrusion and scope intrusion. We deal only with scope extrusion since scope intrusion is similar. In the case of resource names, we have the following typing:

$$\frac{\dfrac{\Gamma + r : s \vdash P : \Delta \qquad r \notin fn(\Gamma)}{\Gamma \vdash \nu r : s.P : \Delta}\ [\text{NU.RES}] \qquad set(\Delta)}{\Gamma \vdash \epsilon[\nu r : s.P] : \Delta}\ [\text{TOP}]$$

with additional conditions on s which we do not detail. Since $set(\Delta)$ we can apply rule TOP to yield $\Gamma + r : s \vdash \epsilon[P] : \Delta$, and, since $r \notin fn(\Gamma)$, we can apply rule TOP.NU.RES to yield $\Gamma \vdash \nu r : s.\epsilon[P] : \Delta$, as required.

In the case of cell names, we have the following typing:

$$\dfrac{\dfrac{\Gamma + a : \mathtt{dom} \vdash P : \Delta \qquad a \notin (\Delta - a) \qquad a \notin fn(\Gamma)}{\Gamma \vdash \nu a.P : \Delta - a}\ [\text{NU.DOM}] \qquad set(\Delta - a)}{\Gamma \vdash \epsilon[\nu a.P] : \Delta - a}\ [\text{TOP}]$$

Since $set(\Delta - a)$ and $a \notin \Delta - a$, we have $set(\Delta)$, and we can apply rule TOP to get $\Gamma + a : \mathtt{dom} \vdash \epsilon[P] : \Delta$. Since $a \notin fn(\Gamma)$, we can apply rule TOP.NU.DOM to get $\Gamma \vdash \nu a.(\epsilon[P]) : \Delta - a$, as required.

STRUCT.NU.MEM We consider the equivalence $a(\nu n.P)[Q] \equiv \nu n.a(P)[Q]$, with $n \neq a$ and $n \notin \mathsf{fn}(Q)$.

Again we distinguish between resource names and cell names, scope extrusion and scope intrusion. We deal only with scope extrusion since scope intrusion is similar (using lemma 4.1.9 instead of lemma 4.1.8). In the case of resource names, we have the following typing:

$$\dfrac{\dfrac{\Gamma + r : s \vdash P : \Delta_1 \qquad r \notin \mathsf{fn}(\Gamma)}{\Gamma \vdash \nu r : s.P : \Delta_1}\ [\text{NU.RES}] \qquad \Gamma \vdash a : \mathtt{dom} \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash a(\nu r : s.P)[Q] : \Delta_1, \Delta_2, a}\ [\text{DOM}]$$

Since $r \notin \mathsf{fn}(\Gamma)$, we can apply Lemma 4.1.8 to get $\Gamma + r : s \vdash a : \mathtt{dom}$ and $\Gamma + r : s \vdash Q : \Delta_2$. We can then apply rule DOM to yield: $\Gamma + r : s \vdash a(P)[Q] : a, \Delta_1, \Delta_2$. Finally, since $r \notin \mathsf{fn}(\Gamma)$, we can apply rule NU.RES to get $\Gamma \vdash \nu r : s.a(P)[Q] : a, \Delta_1, \Delta_2$, as required.

In the case of cell names, we have the following typing:

$$[\text{NU.DOM}]\ \dfrac{\begin{array}{c}\Gamma + b : \mathtt{dom} \vdash P : \Delta_1 \qquad b \notin \mathsf{fn}(\Gamma) \qquad b \notin \Delta_1 - b \\ \hline \Gamma \vdash \nu b.P : \Delta_1 - b \\ \vdots \qquad\qquad \Gamma \vdash a : \mathtt{dom} \qquad \Gamma \vdash Q : \Delta_2 \end{array}}{\Gamma \vdash a(\nu b.P)[Q] : a, \Delta_1 - b, \Delta_2}\ [\text{DOM}]$$

In the above, notice that, through appropriate renaming (using Lemma 4.1.5), we can assume $b \notin \Delta_2$. Since $b \notin \mathsf{fn}(\Gamma)$, we can apply Lemma 4.1.8 to get $\Gamma + b : \mathtt{dom} \vdash a : \mathtt{dom}$ and $\Gamma + b : \mathtt{dom} \vdash Q : \Delta_2$. Applying rule DOM we get: $\Gamma + b : \mathtt{dom} \vdash a(P)[Q] : a, \Delta_1, \Delta_2$. Since $b \notin a, \Delta_2$ and $b \notin \Delta_1 - b$, we have $b \notin (\Delta_1, \Delta_2, a) - b = a, \Delta_1 - b, \Delta_2$. We can now apply rule NU.DOM to get: $\Gamma \vdash \nu b.(a(P)[Q]) : a, \Delta_1 - b, \Delta_2$, as required.

STRUCT.NU.PLASM This case is handled in the same way as the case of rule STRUCT.NU.MEM above.

STRUCT.α This case is an immediate consequence of lemmas 4.1.5 and 4.1.6, considering only renaming to fresh names, and using the version of lemma 4.1.6 with no type inclusion.

STRUCT.CONTEXT This case is immediate by induction and by lemma 4.1.6 with no type inclusion.

□

Lemma 4.2.2 (Substitution lemma) *If* $\Gamma + x : \sigma \vdash P : \tau$ *and* $\Gamma \vdash V : \sigma'$ *with* $\sigma' \subseteq \sigma$*, then* $\Gamma \vdash P\{^V/_x\} : \tau'$ *with* $\tau' \subseteq \tau$*.*

Proof: By induction on the typing derivation of $\Gamma + x : \sigma \vdash P : \tau$, extending the property to definitions.

NIL,VOID Immediate.

NAME If x is u, the result is immediately the hypothesis $\Gamma \vdash V : \sigma'$. Otherwise, the result is immediate (removing the binding $x : \sigma$ from Γ leaves the binding for u in place).

ADDR Immediate by induction (as only $\mathtt{dom} \subseteq \mathtt{dom}$).

FUN By hypothesis of rule FUN, we know that the substituted variable is different from the λ bound variable. To apply the induction, we extend the typing $\Gamma \vdash V : \sigma'$ using lemma 4.1.8 to include the λ bound variable. We conclude by induction, since $\tau' \subseteq \tau \implies \sigma \to \tau' \subseteq \sigma \to \tau$.

DOM Immediate by induction, as x cannot be a (otherwise, this lemma would be much more complicated as the type would also change).

PAR Immediate by induction.

NU.RES Immediate by induction, as r cannot be x. It is also necessary in this case to use lemma 4.1.8 to add the binding for r in the typing $\Gamma \vdash V : \sigma'$ in order to apply the induction hypothesis.

NU.DOM Identical to the NU.RES case. The condition on Δ_1 is preserved through type inclusion.

PASS We immediately have by induction $\Gamma \vdash V'\{^V/_x\} : (\mathtt{unit} \to \rho_1) \to (\mathtt{unit} \to \rho_2) \to \Delta'$ with $\Delta' \subseteq \Delta$. We conclude by rule PASS.

APP By induction we have $\Gamma \vdash P\{^V/_x\} : \sigma \to \tau'$ with $\tau' \subseteq \tau$, and $\Gamma \vdash Q\{^V/_x\} : \sigma''$ with $\sigma'' \subseteq \sigma'$. We may apply rule APP to conclude.

TEST Immediate by induction, since type inclusion is preserved by the "$\wedge$" operator.

DEF Immediate by induction.

AND Immediate by induction.

DEF.$\perp$ Immediate.

JOIN Immediate by induction as x may not be a r_i, and as x is by hypothesis of the rule different from all the $\widetilde{x_i}$. We once again need to use lemma 4.1.8 to extend the typing $\Gamma \vdash V : \sigma'$ with the bindings for the $\widetilde{x_i}$.

□

Theorem 1 (Subject reduction) *If* $\Gamma \vdash \mathcal{S} : \Delta$ *and* $\mathcal{S} \to \mathcal{S}'$*, then there exists* Δ' *such that* $\Delta' \subseteq \Delta$ *and* $\Gamma \vdash \mathcal{S}' : \Delta'$*. The same property is true for processes.*

Proof: By induction on the reduction.

RED.BETA We consider the reduction: $(\lambda x.P)V \to P\{^V/_x\}$

The typing derivation for the initial term is:

$$\frac{\dfrac{\Gamma + x : \sigma \vdash P : \tau \qquad x \notin fn(\Gamma)}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\,[\text{FUN}] \qquad \Gamma \vdash V : \sigma' \qquad \sigma' \subseteq \sigma}{\Gamma \vdash (\lambda x.P)V : \tau}\,[\text{APP}]$$

We apply the subtitution lemma 4.2.2, to yield $\Gamma \vdash P\{^V/_x\} : \tau'$ with $\tau' \subseteq \tau$.

RED.IF.THEN We consider the reduction: $([n = n]P, Q) \to P$

The typing derivation for the initial term is:

$$\frac{\Gamma \vdash P : \tau_1 \qquad \Gamma \vdash Q : \tau_2}{\Gamma \vdash ([n = n]P, Q) : \tau_1 \wedge \tau_2}\,[\text{TEST}]$$

Thus we have the typing: $\Gamma \vdash P : \tau_1$ with $\tau_1 \subseteq \tau_1 \wedge \tau_2$ by proposition 4.1.1(4).

RED.IF.ELSE We consider the reduction: $([n = V]P, Q) \to Q$ with $n \neq V$.

The typing derivation for the initial term is:

$$\frac{\Gamma \vdash P : \tau_1 \qquad \Gamma \vdash Q : \tau_2}{\Gamma \vdash ([n = V]P, Q) : \tau_1 \wedge \tau_2}\,[\text{TEST}]$$

Thus we have the typing $\Gamma \vdash Q : \tau_2$ with $\tau_2 \subseteq \tau_1 \wedge \tau_2$ by proposition 4.1.1(4) and the symetry of "$\wedge$".

RED.PASSIV We consider the reduction:

$$a(\mathtt{pass}_a\ V \mid P)[Q] \to V(\lambda.P)(\lambda.Q)$$

The typing derivation for the initial term is necessarily:

$$[\text{PAR}]\ \frac{\dfrac{\Gamma \vdash V : (\mathtt{unit} \to \rho_1) \to (\mathtt{unit} \to \rho_2) \to \Delta}{\Gamma \vdash \mathtt{pass}_a\ V : \Delta_1}\,[\text{PASS}] \qquad \Gamma \vdash P : \Delta_P}{\begin{array}{c}\Gamma \vdash (\mathtt{pass}_a\ V \mid P) : \Delta_1, \Delta_P \\ \vdots \qquad\qquad \Gamma \vdash a : \mathtt{dom} \qquad \Gamma \vdash Q : \Delta_Q \\ \hline \Gamma \vdash a(\mathtt{pass}_a\ V \mid P)[Q] : a, \Delta_1, \Delta_P, \Delta_Q\end{array}\,[\text{DOM}]}$$

where ρ_1, ρ_2 do not occur in Γ nor in V, nor in Δ_1, and where $\Delta_1 = \Delta - (a, \rho_1, \rho_2)$. In the above, we can always choose, by Lemma 4.1.7, ρ_1 and ρ_2 such that they do not occur in Δ_P nor in Δ_Q. Thus $\{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\}\{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\} = \{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\}$, and we may apply lemma 4.1.7, to get:

$$\Gamma \vdash V : (\mathtt{unit} \to \Delta_P) \to (\mathtt{unit} \to \Delta_Q) \to \Delta\{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\}$$

By two applications of rule APP and of rule FUN, we then get:

$$\Gamma \vdash V(\lambda.P)(\lambda.Q) : \Delta\{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\}$$

Since $\Delta_1 = \Delta - (a, \rho_1, \rho_2)$, and $\rho_1, \rho_2 \notin \Delta_1$, we get $\Delta\{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\} \subseteq (\Delta_1, a, \rho_1, \rho_2)\{^{\Delta_P;\Delta_Q}/_{\rho_1;\rho_2}\} = \Delta_1, a, \Delta_P, \Delta_Q$, as required.

RED.RES We have the following reduction:

$$\langle D\rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} \to \langle D\rangle \mid P\{\widetilde{V_i}/\widetilde{x_i}\}$$

with $\langle D\rangle = \langle D_0; r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} = P\rangle$.

The typing for the initial term is necessarily:

$$\dfrac{\dfrac{[\text{AND}]\ \dfrac{(*)}{\Gamma \vdash D}}{[\text{DEF}]\ \dfrac{}{\Gamma \vdash \langle D\rangle : \emptyset}} \qquad \Gamma \vdash r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} : \Delta}{\Gamma \vdash \langle D\rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} : \Delta}\ [\text{PAR}]$$

To conclude, we only need to replace the judgement $\Gamma \vdash r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} : \Delta$ with the judgement $\Gamma \vdash P\{\widetilde{V_i}/\widetilde{x_i}\} : \Delta_0$, where $\Delta_0 \subseteq \Delta$.

To this end, we rely on the typing of the join pattern:

$$\dfrac{\begin{array}{c}(r_i : s_i = \forall\widetilde{\alpha_i}\widetilde{\rho_i}.\widetilde{\sigma_i} \to \Delta_i' \in \Gamma)^{i\in[1..n]} \\ \Delta' \subseteq \Delta_1', \ldots, \Delta_n' \qquad \Gamma + \widetilde{x_1} : \widetilde{\sigma_1} + \ldots + \widetilde{x_n} : \widetilde{\sigma_n} \vdash P : \Delta' \\ (\widetilde{x_i})^i \cap fn(\Gamma) = \emptyset \qquad \forall i \in [1..n].fv(\widetilde{\sigma_i} \to \Delta_i) \cap fv(\Gamma) \cap (\widetilde{\alpha_i} \cup \widetilde{\rho_i}) = \emptyset \\ \forall i,j \in [1..n]^2.i \neq j \implies fv(\widetilde{\sigma_i} \to \Delta_i) \cap fv(\widetilde{\sigma_j} \to \Delta_j) \cap (\widetilde{\alpha_i} \cup \widetilde{\rho_i} \cup \widetilde{\alpha_j} \cup \widetilde{\rho_j}) = \emptyset\end{array}}{(*)}\ [\text{JOIN}]$$

and on the typing of each message:

$$[\text{NAME}]\ \dfrac{\dfrac{\forall\widetilde{\alpha_i}\widetilde{\rho_i}.\widetilde{\sigma_i} \to \Delta_i' \in \Gamma \qquad \sigma_i^r \to \Delta_i^r = Inst(\forall\widetilde{\alpha_i}\widetilde{\rho_i}.\widetilde{\sigma_i} \to \Delta_i')}{\Gamma \vdash r_i : \sigma_i^r \to \Delta_i^r} \qquad \Gamma \vdash \widetilde{V_i} : \sigma_i'^r}{\Gamma \vdash r_i\widetilde{V_i} : \Delta_i^r}\ [\text{APP}]$$

where $\sigma_i'^r \subseteq \sigma_i^r$.

We have $\Delta = \Delta_1^r, \ldots, \Delta_n^r$, by successive applications of the rule PAR on the messages.

We suppose that the generalized variables are different from all variables occurring in the types $\sigma_i^r \to \Delta_i^r$ (renaming them if necessary to fresh variables, using lemma 4.1.7 on the typing of the guarded process). Let θ_i be the instantiation used in the typing of the resource name r_i. Because of the second generalization condition of rule JOIN, no generalized variable may be shared between two types, thus the domains of the θ_i are disjoint. Let θ be the composition of all the θ_i (the previous condition insures that the order of the composition does not matter). Since all generalized variables are different from variables occurring in the instantiated types, we have $\theta\theta = \theta$.

We now use this substitution on the typing of the guarded process, to yield through lemma 4.1.7 the judgement:

$$\Gamma + \widetilde{x_1} : \sigma_1^r + \ldots + \widetilde{x_n} : \sigma_n^r \vdash P : \Delta'\theta$$

We have:

$$\Delta'\theta \subseteq \Delta_1'\theta, \ldots, \Delta_n'\theta = \Delta_1^r, \ldots, \Delta_n^r = \Delta$$

Applying n times the substitution lemma 4.2.2, to yield:

$$\Gamma \vdash P\{\widetilde{V_i}/\widetilde{x_i}\} : \Delta''$$

(the order of the substitution does not matter as no x_i occurs in Γ, thus they cannot occur in any V_i) where $\Delta'' \subseteq \Delta'\theta$.

We conclude by rule PAR, yielding the smaller type Δ''.

RED.CONTEXT We consider the reduction: $\mathbf{E}\{P\} \to \mathbf{E}\{Q\}$ where $P \to Q$.

Since we have a typing $\Gamma \vdash \mathbf{E}\{P\} : \Delta$, we split this typing into $\Gamma' \vdash P : \tau$ and $\Gamma \vdash \mathbf{E}\{\cdot : \tau\} : \Delta$, with Γ' being the type environment at the typing of the hole. By induction, we have a typing $\Gamma' \vdash Q : \tau'$ with $\tau' \subseteq \tau$. We apply lemma 4.1.6 to yield: $\Gamma \vdash \mathbf{E}\{Q\} : \Delta'$ with $\Delta' \subseteq \Delta$.

RED.TOP.EQUIV We consider the reduction: $\mathcal{S}_1 \to \mathcal{S}_2$ where $\mathcal{S}_1 \equiv \mathcal{S}_1'$, $\mathcal{S}_1' \to \mathcal{S}_2'$, and $\mathcal{S}_2' \equiv \mathcal{S}_2$.

We have the following typing derivation: $\Gamma \vdash \mathcal{S}_1 : \Delta_1$. By lemma 4.2.1 we have the typing derivation: $\Gamma \vdash \mathcal{S}_1' : \Delta_1$. By induction we have the typing derivation: $\Gamma \vdash \mathcal{S}_2' : \Delta_2$. with $\Delta_2 \subseteq \Delta_1$. We conclude applying once again lemma 4.2.1 to yield: $\Gamma \vdash \mathcal{S}_2 : \Delta_2$.

RED.PROC.EQUIV This case is similar to the case of rule RED.TOP.EQUIV above.

ROUTING All reductions immediately have the subject reduction property. The only check is about special channels, we detail the **i** case. We prove that if $\Gamma \vdash r\widetilde{V} : \Delta$, then $\Gamma \vdash \mathbf{i}(\lambda.r\widetilde{V}) : \Delta$. We simply build the derivation:

$$[\text{NAME}]\ \frac{\mathbf{i} : \forall\rho.(\mathtt{unit} \to \rho) \to \rho \in \Gamma}{\Gamma \vdash \mathbf{i} : (\mathtt{unit} \to \Delta) \to \Delta} \qquad \frac{\Gamma \vdash r\widetilde{V} : \Delta}{\Gamma \vdash \lambda.r\widetilde{V} : \mathtt{unit} \to \Delta}\ [\text{FUN}]}{\Gamma \vdash \mathbf{i}(\lambda.r\widetilde{V}) : \Delta}\ [\text{APP}]$$

□

Definition 4.2.3 (Failure) *We say a top-level configuration $\mathcal{S}$ has failed when it contains two active cells with the same name, i.e. $\exists a,\ a, a \subseteq \mathsf{cells}(\mathcal{S})$.*

Theorem 2 (Progress) *If $\Gamma \vdash \mathcal{S} : \Delta$, then $\mathcal{S}$ has not failed.*

Proof: We first prove that we have $\mathsf{cells}(\mathcal{S}) \subseteq \Delta$, by an immediate induction on the typing derivation of the soup. Since by lemma 4.1.3 we have $set(\Delta)$, there cannot be two active cells with the same name. □

Chapter 5

Conclusion

We have presented the M-calculus, a new process calculus with a *cell* construct which provides the ability to capture several interesting features of mobile, distributed programming. Referring to the requirements introduced in chapter 1, we can see that we have obtained a reasonable coverage:

- The calculus includes a primitive notion of cell (*Requirement 2*), which can be understood as a form of superimposition construct $a(P)[Q]$ and which provides the means to partition a distributed computation into asynchronously communicating units (*Requirement 1*).
- Cell membranes in the calculus are first class processes that can send, receive and process messages (*Requirement 3*), as well as create new cells and move all or part of a cell plasm from one cell to another, as explained in section 3.3 (*Requirement 5*).
- The calculus also provides a limited form of message interception in its routing rules (*Requirement 3*).
- The calculus relies on simple directed asynchronous point to point communication, with unique cell names providing the basis for routing Such a calculus can be readily implemented (*Requirement 1*) as demonstrated by the implementation of the distributed join-calculus [13]. The only problematic primitive with respect to implementation is the pass construct, which can be implemented by a simple (possibly recursive) pickling and unpickling of process states, no more complex than the one required for the implementation of the go construct in the distributed join calculus.

Despite a relatively good coverage of the requirements identified in chapter 1, at least compared to other distributed process calculi, several issues warrant further study:

- The type system described in section 2.3 is too simple for actual programming purposes. Extending it with dynamic typing (or any other solution) to handle interception and filtering in a type safe manner, as discussed in section 3.5, is an important requirement. Note that in a practical distributed programming language, dynamic typing would be required anyway to deal with situations where type safe transformations of data are required (e.g. as in marshaling and unmarshaling).
- Even in the simple examples presented in this report, one can detect the presence of an object-oriented style of programming, with cell names playing the role of object identities, and definitions associated with the cell membrane or plasm playing the role of methods. This in turn suggests two questions: is it possible to introduce types for processes to characterize their overall behavior; and is it possible to faithfully encode typed object calculi with the M-calculus ? The different directions suggested e.g. in [13] for object-orientation and the join calculus are directly relevant, mutatis mutandis, for the M-calculus.

- Semantic issues in relation with the M-calculus need to be investigated, including bisimulation semantics, notions of observation and testing, and semantic models in relation with cells.
- Much work is currently taking place on component-based programming. In the light of the discussion in section 3.3, it might be interesting to study the relations between notions of components and the notion of cell as proposed in the M-calculus.

Bibliography

[1] R. Amadio : "An asynchronous model of locality, failure, and process mobility" – Research Report RR-3109, INRIA, Sophia-Antipolis, France, 1997.

[2] G. Berry, G. Boudol : "The chemical abstract machine" – Theroretical Computer Science, vol. 96, 1992.

[3] G. Boudol : "Notes on algebraic calculi of processes" – in Logics and Models of Concurrent Systems, K. Apt (ed.), NATO ASI Series F vol. 13, Springer Verlag, 1985.

[4] G. Boudol : "The π-Calculus in Direct Style" – Higher-Order and Symbolic Computation, vol. 11, 1998.

[5] G. Boudol, A. Schmitt, J.B. Stefani : "Marvel programming model v1" – Deliverable D2.1, Marvel RNRT Project, INRIA, February 2001.

[6] L. Cardelli, A. Gordon : "Mobile Ambients" – Foundations of Software Science and Computational Structures, Maurice Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378, Springer, 1998.

[7] L. Cardelli : "Wide Area Computation" – in Proc. Automata, Languageseand Programming, 26th International Colloquium, (ICALP'99), J. Wiedermann, P. van Emde Boas, M. Nielsen (eds), Lecture Notes in Computer Science, Vol. 1644, Springer, 1999.

[8] I. Castellani : "Process Algebras with Localities" – in Handbook of Process Algebra, J. Bergstra, A. Ponse and S. Smolka (eds), Elsevier, 2001.

[9] N. De Nicola, G.L. Ferrari, R. Pugliese : "Programming Access Control: The KLAIM Experience" – in Proceedings CONCUR 2000, Lecture Notes in Computer Science 1877, Springer, 2000.

[10] C. Fournet, G. Gonthier : "The reflexive chemical abstract machine and the join-calculus" – In proceedings 23rd ACM Symposium on Principles of Programming Languages (POPL), 1996.

[11] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, D. Remy: "A calculus of mobile agents" – in Proceedings CONCUR '96, LNCS 1119, Springer Verlag, 1996.

[12] C. Fournet, C. Laneve, L. Maranget and D. Rémy "Implicit Typing à la ML for the join-calculus" – in Proc. 8th International Conference on Concurrency Theory (CONCUR '97), LNCS 1243, Springer Verlag, 1997.

[13] C. Fournet : "The Join-Calculus" – PhD Thesis, Ecole Polytechnique, Palaiseau, France, 1998.

[14] C. Fournet, J.J. Levy, A. Schmitt: "An Asynchronous Distributed Implementation of Mobile Ambients" – Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, LNCS 1872, 2000.

[15] N. Francez, I. Forman : “Interacting Processes : A multiparty approach to coordinated distributed programming” – Addison-Wesley, 1996.

[16] M. Hennessy, J. Riely : “Resource access control in systems of mobile agents” – Technical Report 2/98, School of Cognitive and Computer Sciences, University of Sussex, UK.

[17] F. Levi, D. Sangiorgi : “Controlling interference in Ambients” – in Proceedings 27th Annual ACM Symposium on Principles of Programming Languages (POPL 2000), Boston, Massachusetts, USA, 2000.

[18] R. Milner : “Communicating and mobile systems : the π-calculus” – Cambridge University Press, 1999.

[19] Object Management Group: “The Common Object request Broker: Architecture and Specification” – Revision 2.4.2, Object Management Group, 2001.

[20] Object Management Group: “CORBA Components” – OMG document orbos/99-02-01, 1999.

[21] A. Schmitt, J.B. Stefani: “The Marvel programming model: a higher-order distributed process calculus”. Deliverable D2.2, Marvel RNRT project, INRIA, November 2001. Available at : http://pauillac.inria.fr/ aschmitt/publications.html

[22] P. Sewell, P. Wojciechowski, B. Pierce : “Location-independent communication for mobile agents : a two-level architecture” – Tech. Report 462, Computer Lab, University of Cambridge, Cambridge, UK, 1998.

[23] J.B. Stefani, F. Germain, E. Najm : “Elements of an object-based model for distributed and mobile computation” – in Proceedings 4th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2000), Stanford, CA, USA, 2000.

[24] Sun Microsystems: “Enterprise Java Beans” – Specification v1.0, March 1998.

[25] J. Vitek, G. Castagna : “Towards a calculus of secure mobile computations” – Workshop on Internet Programming Languages, Chicago, Illinois, USA, 1998.

[26] S. Dal Zilio : “Le calcul bleu : types et objets” – PhD Thesis, U. of Nice-Sophia Antipolis, France, 1999.

[27] S. Dal Zilio : “Mobile Processes: a commented bibliography” – URL : http://research.microsoft.com/ sdal/movep.htm

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399